The work was conducted under contract DASW01-94-C-0054, Task T-S5-1266, for the Defense Information Systems Agency. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

PREFACE

This document was prepared by the Institute for Defense Analyses (IDA) under the task order, Object-Oriented Technology Implementation in the DoD. This document relates to a task objective to develop strategies for the implementation of object-oriented technology (OOT) within specific information technology areas within the Department of Defense. This document is one of a set of four reports on OOT implementation. The other reports, focusing on more specialized areas of OOT, are IDA Paper P-3143, *Object-Oriented Programming Strategies for Ada*; IDA Paper P-3144, *Legacy System Wrapping for Department of Defense Information System Modernization*; and IDA Paper P-3145, *Software Reengineering Using Object-Oriented Technology.* All of this work was sponsored by the Defense Information Systems Agency (DISA).

The following IDA research staff members were reviewers of this document: Dr. Edward A. Feustel, Dr. Richard J. Ivanetich, Dr. Reginald N. Meeson, Dr. Judy Popelas, Mr. Clyde G. Roby, and Mr. Glen R. White.

Table of Contents

List of Figures

List of Tables

EXECUTIVE SUMMARY

Purpose

This report provides an overview of object-oriented technology (OOT) techniques and methods that can be applied to information system development and maintenance in the Department of Defense (DoD). In an effort to modernize its information systems, the DoD is transitioning to the use of OOT, which covers a wide range of techniques, tools, and methods, with varying notations and semantics. Since there is such diversity within OOT, the Defense Information Systems Agency has tasked the Institute for Defense Analyses to determine which OOT techniques and methods were better suited to the information system needs of the DoD. This document is the first of a four-volume set that addresses the use of OOT throughout the software lifecycle and as applied to the more specialized problems of the DoD such as migrating legacy systems. Focusing on the use of OOT for system analysis and modeling, this report provides an extensive survey of object-oriented analysis techniques and activities.

Background

OOT consists of a set of methodologies and tools for developing and maintaining software systems using software objects composed of data and operations as the central paradigm. OOT has shown considerable promise for contributing to satisfaction of DoD needs for software reuse, streamlined system development, systems interoperability, and reduction of maintenance and modification costs. In addition, OOT is nearing maturity in several areas, including OO software engineering methodology, OO programming languages, computer-aided software engineering tools, OO database management systems, and OO standards.

Consequently, OOT is positioned to aid the massive migration and reengineering of DoD software systems from many outmoded systems to fewer, modernized, interoperable, less costly systems. Nevertheless, the transition from traditional software technologies to OOT is not trivial; a substantial learning period is required before the benefits of OOT can be realized. This report is intended to facilitate that learning process by providing an intro-

duction on using OOT for developing new and migrational DoD information systems and an analysis of potential issues in the DoD use of OOT.

OOT Within DoD Software Development

Among the four DoD-defined software program lifecycle strategies for automated information systems examined in this report, the "grand design" strategy was found inadequate for much OO development because it excludes any iteration among phases. The DoD-defined "incremental" strategy also contains a barrier to effective OO development because it disallows iteration in determining specifications. The "evolutionary" strategy is the best match for typical OO systems development, although many OO systems may best fit in the "other" lifecycle strategy because they often do not include the incremental delivery of capabilities characteristic of an evolutionary strategy.

OOT may be incorporated at every stage of a software system lifecycle, including all the traditional phases of analysis, design, programming, test, and maintenance. Analysis of alternative methodologies for the OO analysis and OO design phases identified their central activities as object modeling and dynamic modeling. Every OO methodology developed some model of object classes and their features (e.g., attributes, services, and associations). Some type of dynamic modeling of possible courses of events in the operations of the system are also found in any complete OO development methodology. More specifically, interaction (or event trace) diagrams and state-transition diagrams are the commonly preferred formats for dynamic models. Use case (or scenario) analysis, while not universal, has been increasingly recognized as a valuable tool for many aspects of OO development, including requirements analysis, object modeling, dynamic modeling, and testing. However, while functional modeling using dataflow diagrams appears in several methodologies, it was found to be superfluous and best avoided unless such models already exist as part of legacy documentation.

CHAPTER 1. INTRODUCTION

1.1 BACKGROUND

The Department of Defense (DoD) is moving from many outmoded and proprietary automated information systems to a few modernized and less costly systems capable of such features as interoperability. This transition involves the migration and reengineering of legacy systems as well as the development of new automated information systems (AISs), and raises a number of issues around the proposed use of object-oriented technology (OOT). Such issues include the substantial learning period and curve needed to make the transition from traditional software technologies to OOT, and the inadequacy of most of the current DoD-defined lifecycle strategies for software programs. To complicate matters, OOT covers a wide range of techniques, tools, and methodologies, all of which have varying notations and semantics. With such diversity in mind, the Defense Information Systems Agency has tasked the Institute for Defense Analyses (IDA) to determine which OOT techniques and methods were better suited to the AIS needs of the DoD.

1.2 PURPOSE

This report provides an overview of object-oriented technology (OOT) for use in development and maintenance of AISs in the DoD. These information systems range from command, control, communications, computers, and intelligence systems such as the Global Command and Control System to the more narrowly conceived information management systems for domains such as administration, manpower, personnel, medical, contracting, operations, logistics, materiel management, supply, and maintenance.

In general, the report describes how OOT may be used in developing these types of AISs within the constraints of applicable DoD standards, policies, and procedures. In part, it is an introduction to OOT as applicable to DoD information systems, abstracting from existing object-oriented (OO) methodologies all the essential activities in OO development, ranging from requirements analysis to OO programming. A background in OOT is also provided that supports comprehension of issues and choices in OO development that are addressed in a set of IDA companion reports [IDA95b, IDA95c, IDA95d]. It is not intended

to provide an analysis of the merits of using OOT compared with alternatives such as structured analysis and design; this is provided by an earlier report [IDA93a]. Only a brief summary of the reported benefits of OOT is provided in this report to place the discussion here of OOT use within the context of its perceived benefits for DoD systems.

This report targets a broad audience of individuals looking to establish or strengthen a general understanding of OOT for information systems; therefore, it will be of specific interest to DoD software development managers, project managers, technical leads, and software engineers. It is intended to assist DoD software developers in applying OOT to the development of OO systems, whether in new development or in migration from legacy systems.

1.3 APPROACH

The approach taken for the main body of this report involved comparative analysis of a group of popular OO development methodologies for commonalities and trends. OO development is divided into the four traditional development stages: analysis, design, programming, and test. While OO methodologies generally recommend against a single linear traversal of these stages in favor of iterative loops through them, the separate stages and associated activities are usually distinguished. The essential activities of each stage were extracted from the different methodologies, and differences in placement of common activities by different methodologies were noted. This analysis appears in Chapter 3. Detailed discussion of OO programming techniques in Ada (both 83 and 95 versions) is reserved for a companion report [IDA95b].

A different approach was taken to analyzing alternatives for object data management since there is little in the way of distinct OO methodology in this area. Here, the common alternatives for handling the data components of an OO system were examined and their requirements and limitations identified.

New development activities using OOT follow the basic steps of OO analysis, design, and implementation outlined in Chapter 3. System migration involving reengineering or wrapping also requires the basic OO development activities discussed in this report, though they must be supplemented with their own specialized activities. Additional companion reports provide separate detailed discussions of wrapping [IDA95c] and reengineering activities [IDA95d].

1.4 ORGANIZATION OF DOCUMENT

Chapter 2 discusses legacy systems, DoD software needs, the benefits and features of OOT which address these needs, applicable DoD standards, policies, and procedures, and OOT strategy constraints. Chapter 3 provides an overview of the software lifecycle models, describing how OOT may be incorporated in them. Chapter 4 discusses the use of OOT at each specific stage of the lifecycle model. Appendix A provides a detailed discussion and examples of OO analysis categories: requirements analysis, object modeling, dynamic modeling, and functional modeling. Lists of references, glossary, acronyms, and an index are provided at the end of the report.

CHAPTER 2. OOT AND DOD SOFTWARE NEEDS

2.1 LEGACY SYSTEMS

OOT has shown considerable promise for contributing to satisfaction of DoD needs for streamlined system development, systems interoperability, software reuse, and reduction of maintenance and modification costs (as briefly explained later in Section 2.1 on page 5). At the same time, OOT is approaching maturity in many areas such as OO software engineering methodology, OO programming languages, computer-aided software engineering (CASE) tools for object-oriented analysis, design, and implementation, OO databases, and OO standards. One recent milestone in OO maturity was the release of the Ada 95 standard which brings the full capabilities of OO programming to Ada. OOT is well situated to assist in the massive migration and reengineering of DoD information systems from many outmoded, isolated, or proprietary legacy systems to fewer, modernized, interoperable, open systems. In addition, the benefits of OOT can be realized without the constraints of legacy systems within newly developed automated information systems (AISs).

Legacy systems are understood here in the broad sense of any currently operating automated system that incorporates obsolete computer technology, such as proprietary hardware architectures, closed systems, “stovepipe” designs, obsolete programming languages, or obsolete database systems. The process of software migration consists of converting one or more legacy software systems or applications to a modernized one. This may involve changes in hardware, operating systems, and system architecture, in addition to conversions of programming code and databases. Migration may proceed in stages with progressive modernization of more of the legacy system in each stage. The target of a system migration is the intended end result of that migration, and may include new target requirements, hardware, architecture, design, data management, and implementation aspects.

Different strategies and tactics are possible for applying OOT to different types of DoD information system development. We distinguish three different types of software development; these categories differ primarily in the extent to which legacy software contributes to software development, as illustrated in Figure 1.

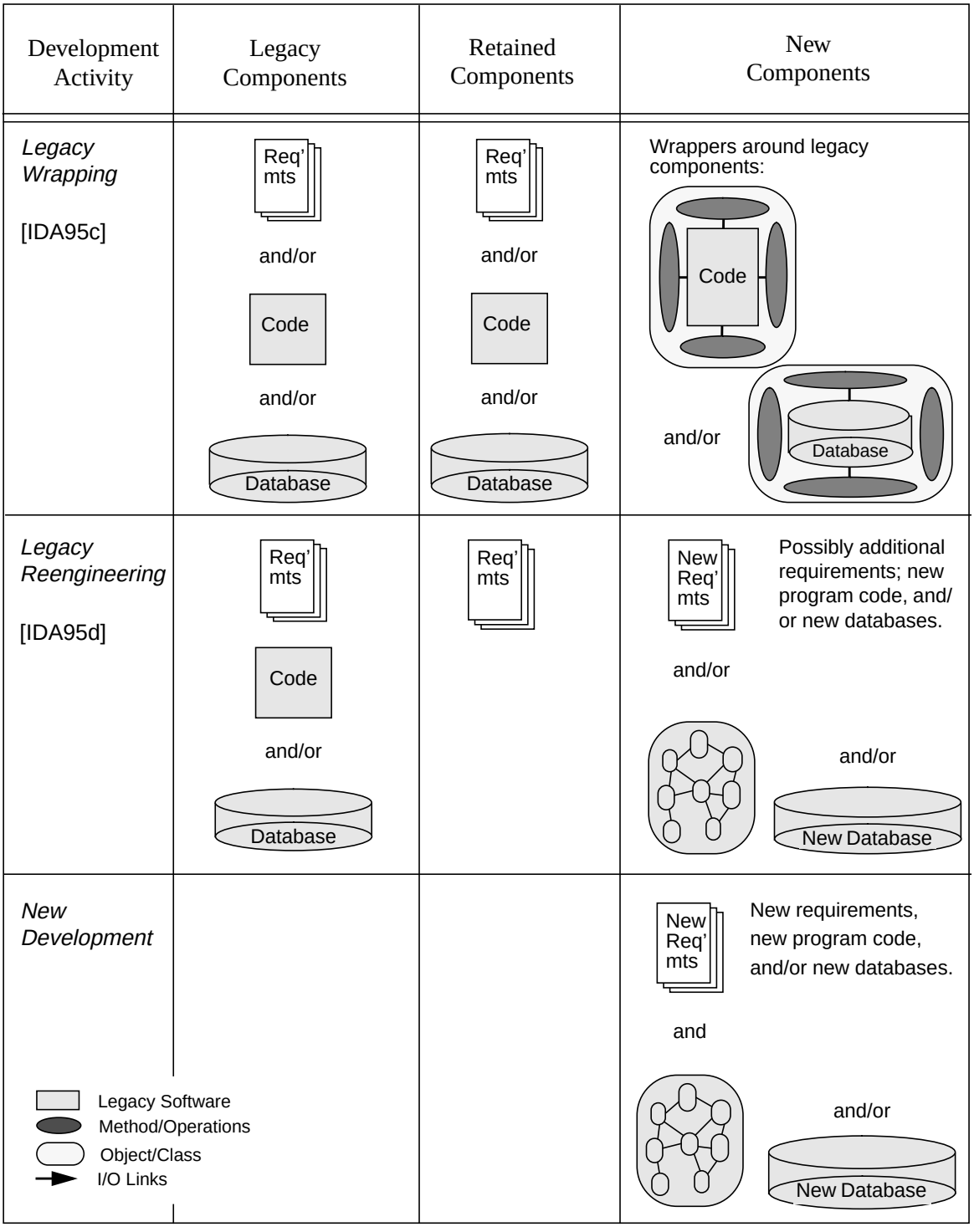


Figure 1. Components of Different Types of Development Activities

- **Legacy Wrapping.** Wrapping of components of legacy systems retains these components within OO wrappers that make them look like software objects within an otherwise modernized OO system. Either legacy code, legacy databases, or both may be wrapped during a particular migration phase.
- **Legacy Reengineering.** Reengineering some part of a legacy system incorporates some legacy requirements, functionality, and possibly some of the legacy data, but recodes all of its software. Reengineered program code is illustrated as an enclosed set of linked ovals to represent the interacting objects of OO programs. Both wrapping and reengineering may be applied simultaneously to different components of a legacy system.
- **New Development.** New development activities are practically independent of any legacy systems, and typically involve both new code and new databases, though they may be limited to one or the other.

2.2 DOD SOFTWARE NEEDS

Recent DoD studies, directives, and policies have given explicit recognition to the pressing need for changes in DoD software, especially in information systems. A plan for Corporate Information Management (CIM) for the DoD, developed in 1990, outlined a vision for DoD information technology in the year 2000 that propounded most of the themes of contemporary DoD information system modernization plans, including the following [DOD90, pp. 10-11]:

- Open systems architectures to accommodate a wide variety of centralized and distributed technologies and products, and of vendor independence.
- Information systems interoperability across the DoD and with allies.
- Reduction of life-cycle costs and development time through increased reuse, reliance on commercial software, use of high-order languages, and improved software development methodologies.

In 1991, the DoD Software Technology Strategy was developed based upon identified features of the changing DoD and world environment that would affect DoD software needs [DOD91, pp. ES-4 - ES-5]:

- Coping with the proliferation of less predictable threats in Eastern Europe and the Third World requires more flexible, interoperable, secure, and reliable C3I, which is software intensive.

- Reduced DoD budgets will increase automation needs to reduce inefficiencies and personnel costs, thus creating further demands on software. Affordability will drive the DoD toward common modular components with flexible software support. This trend holds equally for DoD combat systems, DoD corporate management systems, and manufacturing systems for DoD products.
- A reduced DoD worldwide presence increases the importance of rapid mobilization and deployment which rely critically on software planning and logistics support.
- Reduced DoD human resource levels imply the need for more automated and semi-automated systems to maintain force effectiveness (and to reduce casualty rates). Software is an effective force multiplier for all DoD components.
- Increased networking of DoD systems places escalating strains on computer and software security capabilities.
- Rapid advances in computer hardware capabilities create potential new mission solutions and corresponding software challenges.

Analysis of these conditions and more specific DoD requirements led to the adoption of a software technology strategy driven by five strategic themes [DOD91, pp. ES-6 –ES-7]:

1. Software reuse and megaprogramming - to allow DoD software applications to be developed component-by-component rather than instruction-by-instruction, providing a stronger basis for interoperability.
2. Reengineering and post-deployment software support (PDSS) - will make the huge DoD software inventory easier to support and modify.
3. Process support and technology/management synergy - use process technology improvements to enable software projects to “work smarter.”
4. Commercial technology leverage - with DoD stimulation of commercial technology to ensure support of DoD needs (and DoD use of COTS [commercial off-the-shelf products]).
5. Integration of artificial intelligence and software engineering - to provide new functionality and flexibility from AI with scalability and verifiability of “conventional” software engineering.

These strategies were incorporated into DoD programs, standards, directives, and poli-

cies. DoD Directive 8120.1, for example, states the DoD policy to

> Develop and enhance AISs [automated information systems] in a manner that maximizes the use of standards-based commercial-off-the-shelf (COTS) products, non-developmental item (NDI) products, and commercial items, minimizes the cost of development and the time to deployment, and achieves earliest possible realization of benefits [DOD93a, p. 3].

This is in accord with the strategy of commercial technology leverage. And the instruction for implementing this policy (8120.2) includes a minimum requirement at each phase of the software lifecycle to support the strategy of software reuse, i.e., to "plan for the development and utilization of reusable software assets" [DOD93b, p. 3-4].

The new MIL-STD-498 for software development and documentation provides more specific criteria for evaluation and incorporation of reusable software [DOD92a]:

> 4.2.1. Reusable software. The contractor shall evaluate and incorporate reusable software in accordance with the following requirements . . .
>
> 4.2.2.1 Evaluating reusable software. The contractor shall evaluate reusable software for use in product solutions and in the software engineering and test environments. Criteria for suitability shall be the software's ability to meet user needs and to be cost-effective over the life of the system, including . . .
>
> 4.2.2.2 Incorporating reusable software. The contractor shall incorporate reusable software that, based on the contractor's analysis, will meet user needs and be cost-effective over the life of the system . . .

2.3 OOT FEATURES AND BENEFITS

DoD information system needs are well-served by key features of object-oriented technologies, e.g., to reduce software costs and development time and to improve software reliability.

OOT provides several benefits in meeting the needs identified for DoD AISs. OOT achieves these benefits as a consequence of its fundamental paradigm of software objects and classes of software objects as the basic components of software systems. A software object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class [BOO94a, p. 516]. Software objects represent real-world objects of all types through attributes stored in a set of variables in an object data structure, as illustrated in Figure 2. The attributes of such objects are encapsulated, in the sense that their internal form is not accessible outside the object or class. External information about an object's attributes can only be obtained through the access and update operations that are defined as part of its class. OOT

consists of a full range of software technologies based on such objects, including OO software development methodologies, OO operating systems, OO programming languages, OO programming techniques, OO database architectures, OO distributed programming architectures, object class libraries, object frameworks, and more.

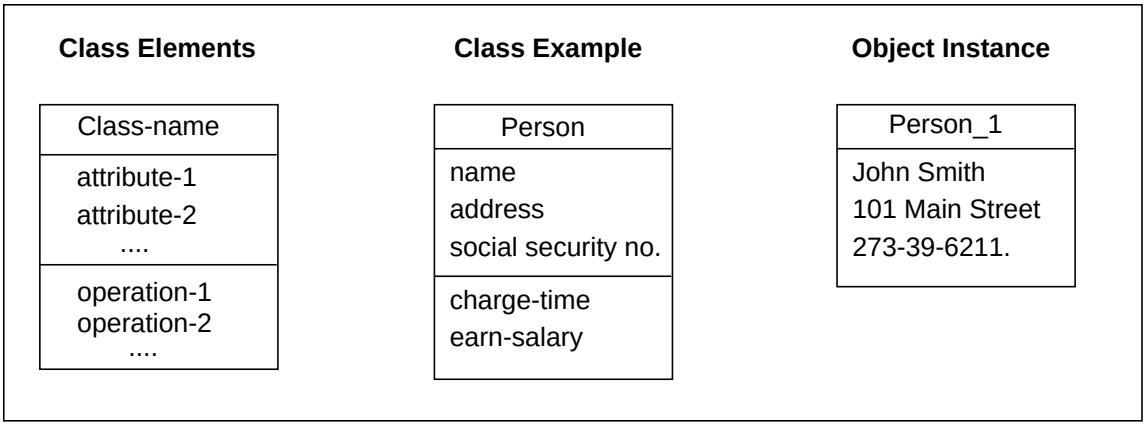


Figure 2. OOT's Objects and Classes

Libraries and repositories of software objects can promote interoperability by establishing standard object classes that are available for use across many different software systems. Standards such as the Common Object Request Broker Architecture (CORBA), developed by the Object Management Group (OMG), have been developed to facilitate distributed access to objects. At a higher level, frameworks, consisting of skeleton structures of many interacting object types, can provide reusable frameworks for applications that share common requirements. Object encapsulation eases maintenance and modification by limiting the "ripple effect" common to modifications in procedural programs, in which modification of one component requires modification of others that interact with it and so on. Object repositories promote reuse to reduce redundancy, so that the same objects and their associated services need not be reimplemented for different systems. Inheritance of class attributes and operations by their subclasses reduces redundancy by allowing features common to multiple object types to appear in just one place in a superclass common to those object types. Capabilities for overriding inherited features facilitate reuse through support of tailoring of repository objects when their features are not a precise match to those required in different applications.

Another OO feature that simplifies development and facilitates reuse is *polymorphism*, in which "a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways" [BOO94a, p. 517]. Development

is simplified because a single function call can replace the case statements and multiple function calls required by non-polymorphic languages. Reuse is facilitated because code using polymorphic operations can easily be extended with new variants of existing classes without changing the calls to these operations. Thus, existing code is more easily extended with new capabilities when polymorphism is utilized. Examples of polymorphism are provided in the separate reports on OO programming in Ada [IDA95b] and on wrapping legacy components [IDA95c].

The reuse encouraged by OOT, in turn, serves to lower development costs once a substantial reuse library has been developed since less software has to be developed from scratch for new (or reengineered) projects. In addition, reuse helps ensure higher quality, more reliable software since library or repository components will ensure extensive testing, both prior to placement in a repository and through field usage on prior projects.

For a thorough assessment of the potential costs and benefits of OOT for DoD information systems, see the earlier study [IDA93a]. The principal costs of OOT use lie in the substantial initial investment required to train development teams in this technology and the resources required to initially establish repositories of reusable object classes and frameworks. The latter costs, however, are faced by any approach to using reusable software. There may also be some performance problems for certain types of database applications when many complex joins are required if an OO database is used. Such database issues are discussed in Chapter 3. For elaboration on the advantages of OOT in specific commercial information system projects, see David Taylor's book, *Object-Oriented Information Systems: Planning and Implementation* [TAYL92]. Other case studies of OO system development can be found in a more recent book by Harmon et al., *Objects in Action: Commercial Applications of Object-Oriented Technologies* [HRTY93].

2.4 OOT STRATEGY CONSTRAINTS

OOT use in DoD is constrained to conform to existing and projected DoD standards, directives, and policies, key elements of which are cited here. Most of these constraints are entirely compatible with a fully OO approach, although, in a few cases, a relaxation or tailoring of a constraint may be required to ensure compliance of strategies with given constraints. Alternatively, an OO design may be modified to accommodate current constraints, for example, the use of SQL for database queries, by storing persistent objects in a relational database instead of an OO database.

2.4.1 MIL-STD-498

MIL-STD-498 [DOD93d], the new standard for DoD software development and documentation, is intended to improve compatibility with non-hierarchical design methods, such as OO design, over the previous standards DOD-STD-2167A and DOD-STD-7935A which it replaces [DOD92a, p. ii]. In particular, it is more supportive of iterative development models that are commonly used in OOT applications, and provides alternatives to, and more flexibility in, preparing documents. MIL-STD-498 was approved as an interim standard for two years in a memo from the Office of the Secretary of Defense (OSD), dated November 8, 1994 [NEWB95]. This standard does not pose any barriers to the use of OOT for developing DoD information systems.

2.4.2 Lifecycle Process Model (DoD 8120)

The DoD Directive (8120.1) and Instruction (8120.2) on Life Cycle Management (LCM) of AISs identify four types of AIS program strategies: grand design, incremental, evolutionary, and "other," a catch-all for other program strategies. Explicit recognition of incremental and evolutionary program strategies has special relevance to OOT-based software programs since experience has shown that large-scale OO software is best managed in an iterative fashion. Thus, this directive can accommodate the most effective life-cycle management of large OO software systems through either of these strategies or the "other" strategy, as explained later in Section 2.2 on page 15.

2.4.3 Functional Process Improvement (8020.1-M)

The DoD Directive (8020.1) and its accompanying Manual (8020.1-M) were established to support the functional (business) process improvement activities of a DoD organization. Functional process improvement is considered a key step before any information system improvement takes place, since improvements to the functional process may have more effect than specific system improvements. DoD 8020.1-M provides specific guidance on establishing mission objectives and on modeling an organization's enterprise-wide functional activities and information requirements. The objective of the activity modeling, which is accomplished with the IDEF0 modeling notation, is to identify "non-value added" activities. These non-value added activities would be eliminated in a revision of the organization's functional process. The objective of the information modeling, which is accomplished with the IDEF1X notation, is to support data element standardization and data administration.

2.4.4 Technical Architecture Framework for Information Management

The Technical Architecture Framework for Information Management (TAFIM) comprises eight volumes of guidance for the evolution of the DoD technical infrastructure. It delineates the services, standards, design concepts, components, and configurations that can be used to guide the development of technical architectures that meet specific mission requirements. It introduces and promotes interoperability, portability, and scalability of DoD information systems [DOD93c, p. 3]. The Director of Defense Information has directed the use of the TAFIM by all new DoD information systems development and major modernization programs [DOD93e].

The guidance provided by the TAFIM is quite broad and accommodating to OOT in general, although there are a few specific areas where aspects of OOT are not fully supported (at this writing). In particular, the only approved standard for database management is SQL which does not directly support OO databases. In the area of programming languages, the Ada 95 standard for OO programming in Ada has not yet been included among programming language standards cited by the TAFIM.

While the TAFIM does not yet incorporate standards for every aspect of OOT, it is intended as "a set of evolving documents" [DOD93e]. Thus, we may reasonably expect that future versions of the TAFIM will incorporate standards supportive of all aspects of OOT, as they become available and as time permits their incorporation. The absence of a complete set of OOT standards within the TAFIM reflects the lack of a consensus on certain aspects of OOT in the broader information processing community, rather than any intentional exclusion of OOT techniques from DoD systems.

More specifically, there is no consensus yet on standards for object-oriented database management systems (OODBMS), although there is work in progress, e.g., by the Object Management Group and in the formulation of SQL-3. In the interim, it is possible to develop an OO database conforming to current data management standards through explicit storage of object data in a relational database management system (RDBMS) with external code completing the support of inheritance, methods, and other object features not available directly in SQL (as discussed further in Chapter 3). This approach has been taken in at least one DoD AIS, the Base-Level System Modernization (BLSM) program [HARR93a]. Alternatively, it may be possible to get an exemption from the Information Technology Policy Board (ITPB) for direct use of a commercial ODBMS, an approach that would address the interests in using a COTS product when it is available.

The current limitations of the Ada 83 programming language standard should not be a constraint on OOT use for much longer because of the new Ada 95 standard which fully supports OO programming. A limited public domain compiler already exists for the core functionality of Ada 95, and commercial compilers are expected shortly. However, fully validated compilers for the full Ada 95 language are not expected for a while since all the validation suites are not even expected until late 1995. In the interim, it is possible to program the functionality of an OO language using Ada 83 (as described in the companion report [IDA95b]), although it can never be as “clean” as Ada 95. Alternatively, an unvalidated Ada 95 compiler may be used for development, provided that an upgrade to a validated compiler is performed prior to the completion of a software development program.

CHAPTER 3. OVERVIEW OF THE SOFTWARE LIFECYCLE

Software development is accomplished in the context of a software lifecycle model that defines the activities, phases, and products of each stage of software development. This chapter reviews the types of software lifecycle models allowable for DoD software development and discusses how the use of OOT fits within these specific models.

3.1 LIFECYCLE PHASES

DoD Instruction 8120.2 on AIS life-cycle management identifies five lifecycle phases:

0. Concept Exploration and Definition Phase - explores alternatives for satisfying the documented mission need and defines the preferred program concept.

    1. Demonstration and Validation Phase - agreement reached on program strategy. Demonstrations and/or rapid prototyping completed and integrated into design. Specifications and design are completed relative to program strategy.

    2. Development Phase - includes development and testing. Activities may be repeated for certain program strategies (e.g., incremental or evolutionary).

    3. Production and Deployment Phase - completes deployment of the AIS.

    4. Operations and Support Phase - operate and maintain AIS, or its increments, evaluate its effectiveness, and plan its modernization.

While there need not be anything particularly object oriented about the concept exploration and definition phase, even for a program committed to doing OO development, some OO techniques might be applied even here. For example, the use case analysis developed by Jacobson [JACO93] for requirements analysis in OO systems could well be used for the essential requirements analysis of this phase.[1] OOT has more extensive contributions to make for subsequent phases of the lifecycle.

---

[1] Use case analysis is described in Appendix A, Section A.1.3.

The demonstrations of phase 1 may be designed, coded, and tested using many of the same OO techniques used in the main development phase. OOT is especially well suited to the rapid prototyping which is optional during this phase. Smalltalk development environments, in particular, have proven very effective for rapid OO prototyping, whether or not Smalltalk is used to implement the delivered system. In addition, phase 1 is where most of the system analysis and design must occur since its minimum required accomplishments include system specifications and design. Thus, OO analysis and design methods, and OO computer-aided software engineering (CASE) tools will play leading roles in this phase of an OO program.

Phase 2, the development phase, is actually focused on implementation under this description of the lifecycle. Note that the development phase of the software lifecycle is often conceived more broadly to include analysis and design, as well as implementation. In any case, during development, the OO modeling of analysis and design is brought to bear on implementation. Ordinarily, implementation of an OO system would use an OO programming language (such as Smalltalk, C++, or Ada95), although other languages can be used with enough effort. OO CASE tools may continue to be of benefit in supplying the OO models and in generating skeletons of object classes in a supported OO language from the models. OO databases or OO techniques for storing objects in relational databases are also likely to be involved in the development phase.

Production and deployment are affected by the information architecture underlying a system, which is partially determined by the basic data representations such as objects or relations. For example, a client-server model supplying information to multiple clients from designated information servers affects the logistics of deployment by its influence on the different types of hardware and software required for different client and server stations. Using OOT in systems development can affect the production and deployment phase through its requirements for support and distribution of various OO capabilities, such as OO databases and OO communication protocols, on deployed platforms. Thus, while there is little, if any, OOT specific to production and deployment, OOT will still influence the execution of this phase of the lifecycle when an OO system is deployed.

The operations and support phase has received special consideration in the OO approach to software engineering. Several of the essential features of OO systems, such as inheritance, encapsulation, and polymorphism, have been designed with the intent of reducing the costs of maintenance during this phase. Maintenance characteristically involves adapting an information system to changing needs, and OO systems are designed to be easier to adapt

with lower risks of breaking existing functionality, as will be described in Section 2.9 on page A-31.

3.2 LIFECYCLE PROGRAM STRATEGIES

DoD Directive 8120.1 and Instruction 8120.2 identify four alternative lifecycle program strategies: grand design, incremental, evolutionary, and "other." The first three are illustrated in Figure 3. More formal descriptions of the alternative strategies are given in the following paragraphs [DOD93b].

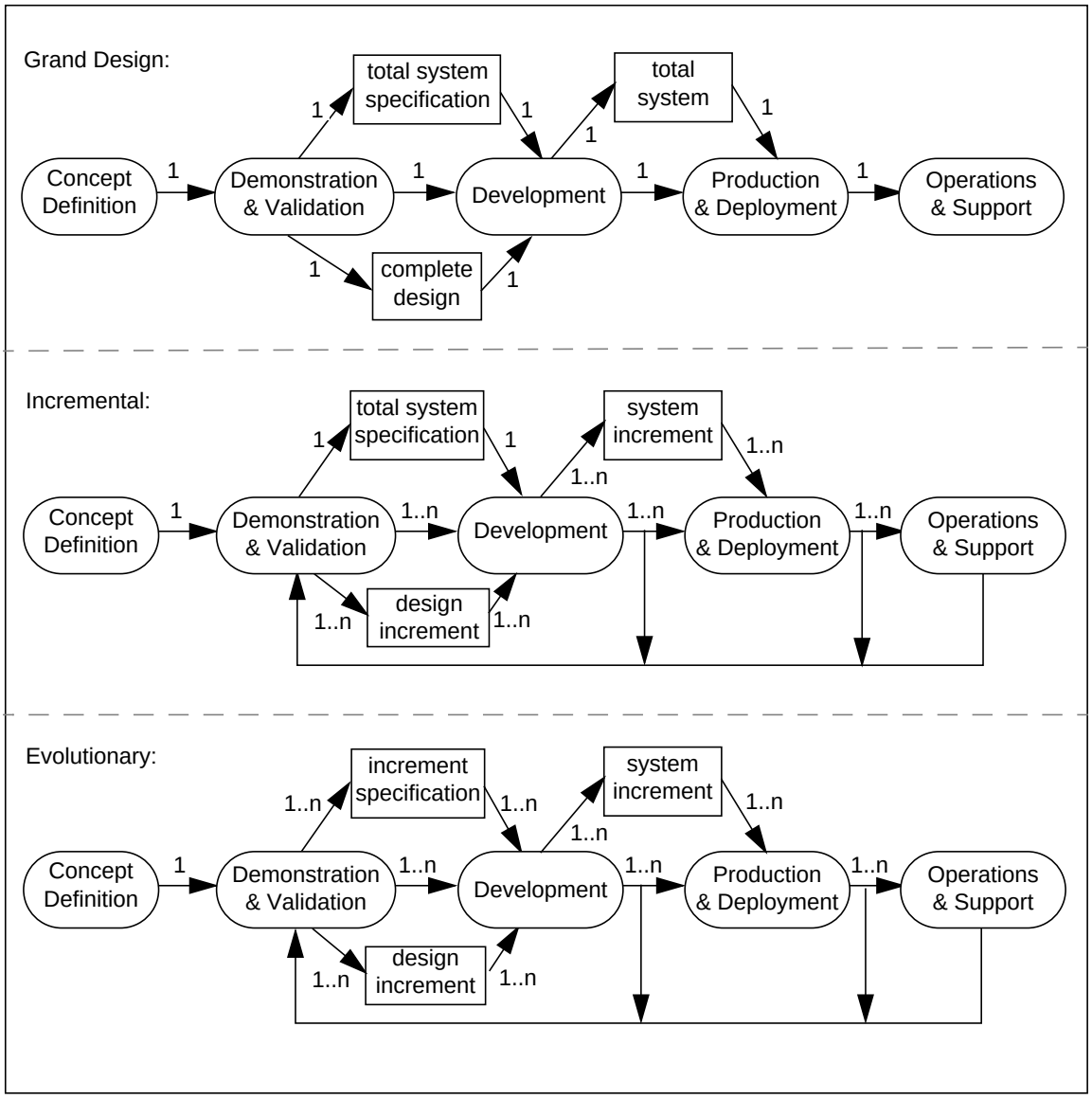


Figure 3. Alternative Program Strategies for System Lifecycle

The *grand design strategy* is characterized by acquisition, development, and deployment of the total functional capability in a single increment. The required functional capability can be clearly defined and further enhancement is not foreseen to be necessary. A grand design program strategy is most appropriate when the user requirements are well understood, supported by precedent, easily defined, and assessment of other considerations (e.g., risks, funding, schedule, size of program, or early realization of benefits) indicates that a phased approach is not required. In short, this strategy simply consists of a single pass through each lifecycle phase with the total specifications of the demonstration and validation phase leading to the total system resulting from development. The numbers on the transition lines between phases (or between a phase and its products) in Figure 3 on page 17 indicate the number (or numbers) of the repetition (or repetitions) at which this transition may occur (or product may be generated).

The *incremental strategy* is generally characterized by acquisition, development, and deployment of functionality through a number of clearly defined system "increments" that stand on their own. The number, size, and phasing of the "increments" required for satisfaction of the total scope of the stated user requirement must be defined by the AIS Program Manager (PM) in consultation with the functional user. An incremental program strategy is most appropriate when the user requirements are well understood and easily defined, but assessment of other considerations (e.g., risks, funding, schedule, size of program, or early realization of benefits) indicates that a phased approach is more prudent or beneficial. In short, an incremental strategy allows for incremental development, although the total system specification must still be completed in the first increment. The demonstration and validation phase may be repeated under an incremental strategy, though the total system specification is not recreated.

The *evolutionary strategy* is generally characterized by the design, development, and deployment of a preliminary capability that includes provisions for the evolutionary addition of future functionality and changes, as requirements are further defined. The total functional requirements the AIS is to meet are successively refined through feedback from previous increments and reflected in subsequent increments. Evolutionary program strategies are particularly suited to situations where, although the general scope of the program is known and a basic core of user functional characteristics can be defined, detailed system or functional requirements are difficult to articulate. The evolutionary program strategy differs from the incremental program strategy because the total functional capability is not completely defined at inception but evolves as the system is built. In short, an evolutionary strategy allows for increments in both the system development and its specification, as illustrated in Figure 3. When there are repeti-

tions of phases, their numbers and the points at which they are initiated should be presented in the proposed program strategy resulting from the concept definition phase.

The *other strategy* encompasses variations and/or combinations of the above program strategies or other program strategies not listed, e.g., the Office of Management and Budget (OMB) Circular-109 acquisitions, commercial-off-the-shelf (COTS) products, nondevelopmental item (NDI), and commercial item acquisitions. The other strategy is a catch-all for other strategies, e.g., those involved in acquiring existing commercial or government software.

These definitions and the accompanying figure have been depicted in brief. For a full description of these program strategies, the reader is referred to DoD Instruction 8120.2.

OOT may be incorporated into software systems fitting into any of the four program strategies. However, many experiences building OO software indicate that a grand design strategy is ill-suited for developing large-scale OO systems. Texas Instruments, for example, moved away from the waterfall model of software development towards an iterative method with prototyping at each stage in its Computer Integrated Manufacturing project with the Air Force and the Advanced Research Projects Agency (ARPA) [TAYL92, p. 323]. Petroleum Information, in its Graphical Information System (GIS) application framework called *Sorcerer's Apprentice*, found that rapid prototyping was more effective in applications development than writing extensive specifications upfront as required by grand design strategies [TAYL92, p. 327]. Brooklyn Union Gas, in its customer management system, used a layered construction of object classes with independent testing at each layer, which is another type of iterative development [TAYL92, pp. 335-338].

The problem with a grand design strategy is that it requires completing both specifications and design during a single demonstration and validation phase prior to any implementation. The minimum required accomplishments of phase 1 in a grand design include the following:

> Detailed specifications are prepared and documented for the total system. The AIS design is complete and based on refined functional requirements, final standards profiles, DoD standard data elements, and the AIS functional description [DOD93b, pp. 3-9].

This is not compatible with the iterative analysis, design, and implementation that are characteristic of typical successful OO projects. However, iteration may be less important for some OO future projects if they are variations upon existing systems built by reusing existing frameworks of class definitions designed to support that type of application. While OO frameworks for applications development are just beginning to be introduced into the current software

market, they may prove to be the basis for much system development in the future. Software development using OO frameworks may be considered a type of grand design strategy when it completes all specifications and design prior to implementation. Under this interpretation, grand design may still be considered a viable strategy for OO development. But using frameworks may better be considered a distinct type of lifecycle strategy since so much of the code exists prior to specifications and design. In any case, outside of such contexts, the current consensus among OO developers is that an iterative strategy is best, if not essential, for large-scale OO development.

Even an incremental strategy, as defined, is difficult to adopt within an iterative OO development because it too requires the total functional capability to be specified during phase 1, whereas specifications are commonly refined throughout OO development. Since an evolutionary strategy allows incremental analysis, design, and implementation, it is the best fit among the three strategies for OO development of large-scale systems. However, the evolutionary strategy has been interpreted to require deploying interim software [DOD94a, p. 47], which may not occur in many OO programs. Hence, the lifecycle of many OO programs may only fit within the "other" category of DoD Instruction 8120.2 strategies.

We conclude that a "grand design" program strategy is a poor candidate for typical large-scale OO systems because of its non-iterative lifecycle, while the DoD-defined "incremental" strategy may be problematic because it disallows iterative development of specifications. Thus, of the specific strategies in DoD Instruction 8120, an evolutionary strategy is the best match for OO systems, although many OO programs may best fit the "other" strategy.

3.3 OBJECT-ORIENTED DEVELOPMENT

While the previously identified program strategies offer one perspective on alternative strategies for software programs, more specific software lifecycle alternatives for OO software engineering have been developed that concentrate on the developmental aspects of software. OO software development is commonly understood to include analysis, design, and implementation [RUMB91, p. 1], as they will be described shortly in this section.

However, this conception of *software development*, which is used throughout the rest of this report, has a different scope than that of the *development phase* of the software lifecycle of DoD Instruction 8120.2. Analysis and design are included in the common conception of OO software development, while they are relegated to the *demonstration and validation phase* for

both grand design and incremental program strategies by DoD Instruction 8120.2. The development phase in the DoD Instruction 8120.2 lifecycle has variable content, depending on the chosen strategy. It is devoted to coding and testing the system or current increment in grand design and incremental strategies, respectively, though it may also include design in an evolutionary strategy. However, these variations in conceptions of *development* simply reflect different uses of the term and in no way indicate any incompatibility between the DoD Instruction and common OO conceptions of the software development lifecycle. OO development, as commonly conceived, will often extend across both the *demonstration and validation phase* and the *development phase* of the DoD Instruction 8120.2 lifecycle. With this clarification, we may proceed to describe characteristic features and alternatives for an OO software development lifecycle without confusion with the broader program lifecycle discussed in Section 3.1.

Practically all OO methods endorse some degree of iteration during development. This endorsement has grown out of the widespread project experience that object models are most naturally constructed in an iterative fashion, since exhaustive analysis of an OO system without the feedback of design and implementation is difficult. Typically, whole new classes, collaborations, responsibilities, and class associations are discovered as analysis, design, and implementation proceed and iterate. Iteration, in this sense, simply indicates that some stages of the development are repeated. A typical OO development model will have a main iteration loop of analysis, design, implementation, test, and maintenance as depicted in Figure 4. Additional iteration loops are ordinarily recommended between other phases, such as a cycle through analysis and design.

In addition to being iterative, an OO software development may be incremental, in the sense of DoD Directive 8120, if the software products of the incremental stages “stand on their own” enough to be deployable. Some OO software methodologies (e.g., the Booch method [BOO94a]) recommend both interactive and incremental development in the sense of yielding deployable increments of functionality at the completion of multiple cycles of iteration. Iteration may occur without being incremental if software is not deployed or deployable between the iterations. This may occur, for example, if iteration is confined to a cycle of analysis and design while implementation is postponed until that cycle has fully converged. Non-incremental iteration also occurs when iteration is at a fine scale, creating increments of software that are not complete enough for deployment.

Some OO software engineering methodologies incorporate novel lifecycle strategies that may be considered to fit best in the “other” category of the DoD’s program lifecycle strategies. The “baseball model” of the Coad-Yourdon methodology is one such example, as illus-

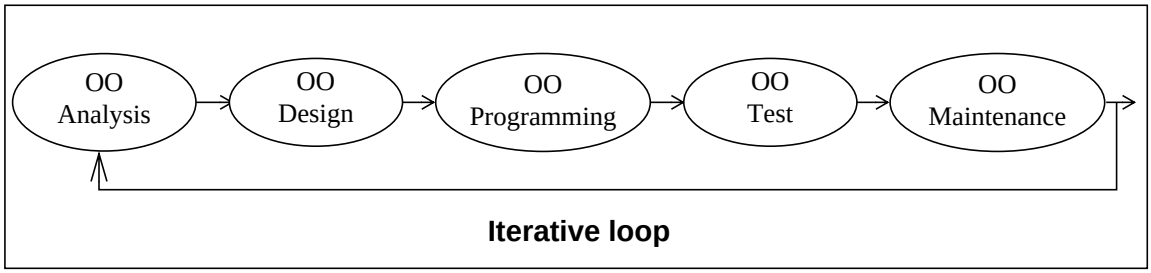


Figure 4. Typical OO Development Iteration

trated in Figure 5. It involves a type of concurrent development in which OO analysis, OO design, and OO programming may all occur concurrently. Planning and control of this development strategy are achieved by the use of "time-boxing," wherein fixed-duration time periods (e.g., four weeks) are identified within which a somewhat integrated set of activities in OO analysis, OOD, and OO programming are completed. The code resulting from these periods of development is not discarded, as it is in prototypes, but is built upon in each successive time box until the system is complete [HUTT94, pp. 44-45]. Since there is no requirement in Coad-Yourdon's approach that the resulting increments should be deployable, consequently it does not fit the incremental or evolutionary strategies of DoD Instruction 8120.2 very well.

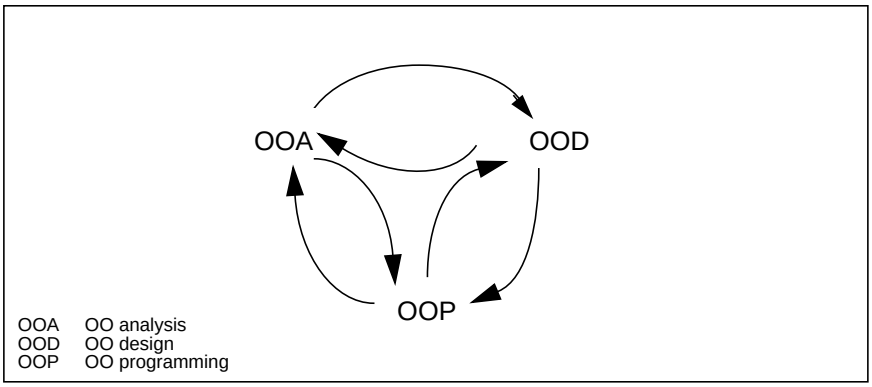


Figure 5. The Baseball Model

While most other methodologies are not so insistent about concurrency in the phases of the development lifecycle, many recognize some overlap or parallel development among them. The Shlaer-Mellor method acknowledges that "there is typically significant overlap between work being done on them" [HUTT94, p. 166]. In the Booch method, a micro development process, at the level of an individual developer or small teams of developers, is identified, in which "the traditional phases of analysis and design are intentionally blurred, and the process is under opportunistic control" [BOO94a, p. 235]. Jacobsen's Objectory methodology "divides devel-

opment into processes that, unlike traditional development phases, can iterate and overlap" [HUTT94, p. 107]. Rumbaugh's Object Modeling Technique (OMT) allows that "some activities may be done in parallel" [HUTT94, p. 157].

> *We conclude that some iteration of the traditional stages of development is a best practice in OO software engineering, which has been found to make large software projects more manageable.*

Our review of OOT throughout the software lifecycle discusses the traditional stages of analysis, design, implementation, testing, and maintenance, with a focus on the developmental stages of analysis, design, and implementation. While OO methodologies do not take a simple linear path through these stages, the different steps, activities, or processes prescribed by OO methodologies can usually be clearly partitioned among them. As we proceed through each stage, the activities from various methodologies that are associated with it will be identified and explained. Some of these activities, such as building an object model during analysis, can be considered essential components of any OO software development, while other activities will be identified as optional techniques peculiar to one or more methodologies. In this way, a composite picture is developed of how OOT may be applied throughout a lifecycle.

CHAPTER 4. OBJECT-ORIENTED DEVELOPMENT ACTIVITES

OOT can be a part of nearly every activity in software development. This section discusses the use of OOT in three activities which range from requirements analysis to maintenance to data management. Detailed discussions about object modeling techniques are provided in Appendix A.

4.1 OBJECT-ORIENTED ANALYSIS

While OO methodologies have some differences in the detailed activities they prescribe at different stages of software development, most of them distinguish the phases of analysis, design, and implementation. For most OO methodologies, OO analysis consists of the following activities: requirements analysis, object modeling, dynamic modeling, functional modeling, and risk analysis. A more detailed discussion of each activity with examples is found in Appendix A.

- **Requirements analysis.** When distinguished as a separate stage of analysis, requirements analysis is always the first one, developing the initial analysis of requirements which provides the rationale for all subsequent analysis. Regardless of whether a chosen OO methodology incorporates requirements analysis, it will be required in any DoD information system development that conforms with MIL-STD-498 [DOD94a]. While OO requirements analysis is typically couched in functional terms, as with non-OO methodologies, the next stage of object modeling is one activity that is distinctively object oriented.
- **Object modeling.** The essence of any OO development, object modeling identifies and structures the objects and classes of the application domain, including their interactions (or collaborations) and associations, such as subclassing and whole-part decomposition.
- **Dynamic modeling.** Widely included in most OO methodologies, dynamic modeling captures the time-varying behavior of objects or classes, usually represented in terms of state transition graphs of some type. Some such dynamic modeling is

essential for effective system design, though it need not be included in the analysis stage.

- **Functional modeling.** Functional modeling describes the flow of information through the system, from external inputs through operations and interactions with data stores to output. Functional models are commonly represented by the data flow diagrams that are central to traditional "structured analysis" methodologies. Several OO methodologies (e.g., Rumbaugh's OMT and the Shlaer-Mellor method) have used data flow diagrams as one means of identifying required class operations, while others have discarded them as being difficult to integrate with an OO perspective (e.g., Coad-Yourdon's [CDYD91] and Booch's [BOO94b] methods). It now appears as though there is a movement away from using data flow diagrams in OO development since even former advocates (e.g., Rumbaugh) have acknowledged that they are difficult to integrate with OO models [BOO94b].
- **Risk analysis.** As a separate analysis activity, risk analysis receives very little attention from OO methodologies since there is nothing particularly object oriented about it, although Booch identifies it as a potential guide to prioritizing subsequent design activities [BOO94a].

4.2 OBJECT-ORIENTED DESIGN

The design phase in OO development transforms the idealized system models of the analysis phase into plans for implementation, establishing a general system architecture that includes the organization of classes into modules and subsystems, environmental fixtures and policies, and revision of the object and dynamic models to accommodate the constraints of the system's architecture, environment, and policies. Although activities in OO design are organized in somewhat different ways by different OO methods, one simple division that applies well to most methods is to be found in OMT's distinction between *system design* and *object design* [RUMB91]. System design includes activities that cover system-wide issues of organization and policy, while object design focuses on the detailed modeling of objects and their features within the context of the system environment. We discuss the activities of each of these parts of design in Sections 4.2.1 and 4.2.2, and describe how they relate to the alternative partitions of the design phase found in some other OO methods.

4.2.1 System Design

While different methodologies may emphasize different activities during system design, they are more notable for their commonalities than their differences. One example of a fixed set of system design activities comes from Rumbaugh's OMT [RUMB91, pp. 198-211], whose steps we may paraphrase as follows:

- Organize system into subsystems.
- Allocate subsystems to processors and tasks.
- Determine policy on object data management.
- Determine policy on access of other global resources.
- Determine software control policy.
- Determine trade-off priority policies (among time, space, and simplicity).
- Establish boundary condition design (initialization, termination, failure).

Other OO design methods typically include these activities, although they may not divide the design phase up neatly between system design and object design.

Booch's method is one example that clearly includes all of Rumbaugh's system design activities. The two primary products of Booch's design phase are an architecture description and a description of common tactical policies. His architectural description includes module diagrams identifying the organization of the system into modules and subsystems, and process diagrams showing the assignment of processes to processors. Thus, the first two steps in Rumbaugh's OMT system design process fit into Booch's architectural planning component of design. The remaining steps of OMT's system design process are all covered by what Booch calls tactical design, where the many common policies for a system are established [BOO94a, pp. 255-257].

Booch's approach to system design differs from OMT in its emphasis on prototyping for design validation and its inclusion of release planning as part of design [BOO94a]. Throughout his discussion of analysis and design, Booch repeatedly advocates rapid prototyping as an aid to a variety of development activities, such as determining requirements (e.g., for user interfaces) and validating system models and architectural designs. In all cases, care is taken to emphasize that prototypes are essentially incomplete abstractions of parts of the system, and prototype code is not intended for incorporation into the final system. Such prototyping contrasts strongly with incremental development in which testable working code is developed

in increments designed to be retained (or evolved) throughout system development into the delivery system. Most OO methodologies, including Booch's method, recognize the proven value of incremental development. Some, like Booch, also emphasize the value of prototyping. In the design phase, Booch suggests using prototypes to validate the overall architecture by satisfying the semantics of several key scenarios as well as to validate specific policies such as data management or control.

Booch's inclusion of release planning in his design phase ventures into pragmatic issues of project management that are commonly separated from basic analysis and design in other methods such as OMT and Jacobson's OOSE (Object-Oriented Software Engineering). Jacobson, for example, devotes a separate chapter to "managing OO software engineering" [JACO93]. Booch himself actually has a separate chapter titled "pragmatics" that is devoted to such management issues as staffing, release management, and quality assurance [BOO94a]. So the inclusion of release planning in his design phase may be considered just one of several pointers he provides to essential management activities throughout his exposition of the development process. Such pointers do not represent any real disagreements with other methods but express an expansion of scope to include some management issues as well as technical ones.

Some disagreement exists between different methodologies on the extent to which the specifics of the implementation environment ought to be considered during the design phase. Rumbaugh maintains that design should

> . . . make the decisions that are necessary to realize a system without descending into the particular details of an individual language or database system [RUMB91, p. 263].

Jacobson, in contrast, clearly includes the effects of these details of the implementation environment in his design phase when he maintains

> The design model will further refine the analysis model in the light of the actual implementation environment. Here . . . we will decide how different issues such as DBMSs, programming language features and distribution will be handled." [JACO93, pp. 204-205]

Other methodologies, such as Booch's method, are not so clear about the extent to which the implementation environment should influence design. In any case, it is clear that many environment-dependent policies about object implementation will be required before coding can proceed. Whether this policy making is labeled as part of design or of implementation matters little, especially within an OO development context where the activities of different phases overlap and the whole process is iterative.

4.2.2 Object Design

Object design may be described as being composed of those design activities that directly elaborate the plans for implementing the components of the object model, including its class structure, operations, attributes, and associations. Object design may be distinguished from system design by its focus on elaborating the design of individual classes, objects, and their features, whereas system design addresses system-wide issues of organization and policy. Object design is distinguished from implementation by the absence of coding which is the principal activity in the next development phase of implementation (or programming). An exception to the exclusion of coding from the design phase occurs when prototypes are used to validate design plans. Coding that is implemented for prototypes may be considered part of design, provided it is not incorporated into the actual system. Otherwise, when coding for validation of design is retained for the delivered system, it is best categorized as part of the implementation phase which may, of course, be interleaved with design in an OO development.

One of the most explicit lists of object design activities is found in Rumbaugh's OMT, which includes the following ordered steps:

- Settle assignment of operations to classes.
- Design algorithms to implement operations.
- Optimize access paths to data.
- Design control strategy.
- Adjust inheritance hierarchy.
- Design associations.
- Determine object representations.
- Package classes and associations into modules.

While these steps fairly represent most of the class-specific steps required to prepare for the transition from an analysis model to actual code, there is some disagreement among the different methods on whether all these activities should be assigned to the design phase. Algorithm design, for example, is included the implementation phase of Booch's "micro-development process." He describes both the "selection" and decomposition of algorithms (when necessary) as part of implementation [BOO94a, pp. 247-248]. Most of the other OMT object design activities are just refinements of the object model that are commonly included in design.

Jacobson departs from OMT and from many other methodologies by delaying until his design phase the initiation of several dynamic modeling activities which others incorporate into analysis, including the following [JACO93]:

- Creation of object interaction diagrams.
- Creation of state transition diagrams.

Jacobson creates a design model based on blocks which are logical-level design objects consisting of one or several classes/objects derived from his analysis model. His design model is composed of the following:

- Block structure diagrams - based upon the analysis object model, modified to accommodate implementation policies and constraints.
- Interaction diagrams - based upon blocks.
- State transition diagrams - for blocks with nontrivial dynamic behavior.

These diagrams are essentially the same as those used by other methodologies during analysis except that the models are further refined to accommodate implementation. Other methodologies, such as OMT, the Shlaer-Mellor method, and the Booch method, that develop dynamic models during analysis prescribe comparable refinements on them during design. Thus, there is very little difference between Jacobson and these other methodologies on what dynamic models are developed, though they assign comparable activities to different development phases.[1] Jacobson is distinctive in his emphasis on switching from the objects/classes of analysis to blocks during design. But this is arguably a negligible difference since blocks are initially mapped one-to-one to objects/classes from his analysis model, and their subsequent evolution is comparable with the evolution that the analysis model makes during design in other methods.

4.3 OBJECT-ORIENTED PROGRAMMING

OO programming is the software development phase where the system models are transformed into actual code in a programming language. When OO analysis and design are thoroughly executed using OO techniques like those discussed in Sections 4.1 and 4.2, coding should be relatively straightforward, consisting principally of translating the object model into code. However, substantial latitude for alternative implementations always exists in such trans-

[1] There are substantive differences in functional modeling, however, as noted in Appendix A's section on functional modeling (Section A.4 on page 48).

lations. The central task of the OO programming phase of software development is the determination of which among the many alternative encodings of an OO design model will actually be implemented. Depending on the programming language used, even the basic implementation of classes, their attributes, and their operations may admit many alternatives. Such alternatives are minimal within strictly OO languages like Smalltalk, increase in languages with fewer built-in OO features, such as Ada 83, and are maximal in languages with no predefined OO features, such as Fortran.

When implementing an OO system, it is most natural to use a fully object-oriented programming language. However, some DoD information systems are committed to using the Ada programming language whose fully object-oriented version (Ada 95) lacks validated compilers at this writing. Thus, DoD information systems programmed using validated Ada compilers will be constrained to use Ada 83 for a while longer. However, this language admits a wide variety of different implementations of class structures and their features.

In a companion report on OO programming strategies for Ada [IDA95b], we examine the principal implementation choices in translating an OO design into an implementation in Ada. Alternative strategies for implementing class hierarchies, inheritance, polymorphism, and associations are described for projects using Ada 83. In addition, we look ahead to the time when Ada 95 will be available in validated compilers and describe the intended usage of its new features for implementing OO systems.

4.4 OBJECT REPOSITORIES

Repositories of object classes, their methods, and their instances participate at both ends of the software development cycle. In the analysis and design phases, a pre-existing object repository may provide actual class and object definitions for an application domain. During the implementation phase, the repository implementation code may then provide the corresponding actual code, supplemented by additional attributes or operations in specializations as needed. At the completion of testing, selected new or modified object classes may be submitted to the object repository for subsequent reuse.

Some methodologies offer guidance on designing classes to optimize their reuse potential. Rumbaugh's OMT, for example, includes the following recommendations for enhancing reusability [RUMB91, pp. 282-283]:

- Keep methods coherent (with a focused functionality).
- Keep methods small (break-up large methods).

- Keep methods consistent (e.g, in argument types between methods).
- Separate policy and implementation methods.
- Generalize methods (cover all relevant conditions).

Separate guidelines have also been provided for enhancing extensibility and robustness of classes, which naturally aid reusability as well. These guidelines include the following [RUMB91, pp. 285-288]:

- Encapsulate classes and data structures.
- Avoid traversing multi-link association paths (from a single method).
- Avoid nested method calls (calling one method with result of another).
- Distinguish public and private operations, attributes, and associations.
- Validate arguments.
- Protect against errors (handle application and system errors gracefully).
- Avoid predefined limits.

Other suggestions on preparing OO software for reuse are found in Jacobson's chapter on components which could be individual classes or frameworks of many classes [JACO93, pp. 289-312]. Such guidelines can aid the development of object repositories with greater reuse potential that substantially reduces the time and costs of subsequent OO development efforts.

4.5 OBJECT-ORIENTED TESTING

Until recently, testing was one phase of software development that had received little attention from OO methodologists. Most of the OO methodology books, such as [BOO94a] and [RUMB91], do not even address testing methodology.

A brief but informative overview of issues in OO testing can be found in the following materials:

- [BARB94] Barbey, S., M. Ammann, and A. Strohmeier, *Open Issues in Testing Object-Oriented Software*, *Technical Report No. 94/45*. Departement D'Informatique, Swiss Federal Institute of Technology—Lausanne, Switzerland, 1994.

A good collection of articles on OO software testing can be found in a special issue of *Communications of the ACM* on this topic from September 1994. This collection consists of the following articles in order of appearance:

- [JORG94] Jorgensen, P. C. and C. Erickson, "Object-Oriented Integration Testing," *Communications of the ACM,* Vol. 37, No. 9, September 1994.
- [BIND94a] Binder, R. V., "Object-Oriented Software Testing: Introduction," *Communications of the ACM,* Vol. 37, No. 9, September 1994.
- [MRPH94] Murphy, G. C., P. Townsend, and P. S. Wong, "Experiences With Cluster and Class Testing," *Communications of the ACM,* Vol. 37, No. 9, September 1994.
- [PSTN94] Poston, R. M., "Automated Testing from Object Models," *Communications of the ACM,* Vol. 37, No. 9, September 1994.
- [MCGR94] McGregor, J. D. and T. D. Korson, "Integrating Object-Oriented Testing and Development Processes," *Communications of the ACM,* Vol. 37, No. 9, September 1994.
- [ARNL94] Arnold, T. R. and W. A. Fuson, "Testing 'In A Perfect World'," *Communications of the ACM,* Vol. 37, No. 9, September 1994.
- [BIND94b] Binder, R. V., "Design for Testability in Object-Oriented Systems," *Communications of the ACM,* Vol. 37, No. 9, September 1994.

In addition, papers on OO testing have appeared in the *Journal of Object-Oriented Programming* and in the published proceedings of the annual conferences on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).

4.6 OBJECT-ORIENTED MAINTENANCE

The modularity of OO technology is designed to minimize disruptions to software systems due to routine software maintenance required by bug fixes, updates, and extensions. The encapsulation of internal data structures characteristic of OO systems, in particular, reduces the risks of change propagation wherein a minor change in one part of a system propagates throughout the entire system, requiring many changes in other subsystems related by chains of interaction. Updates and extensions to the attributes and operations of a class can be made without any disruption to the original class definition by the creation of a new subclass inheriting the original's features and supplementing them with any new features that may be required. Polymorphism allows existing calls to object services to automatically access the new or refined services of new subclasses simply by using the same name for the revised services in the new class. In contrast, conventional programming techniques require modifying case statements for every new variation on a service.

4.7 OBJECT DATA MANAGEMENT

This section discusses databases and files in the framework of OOT. No matter what technology is used to build an AIS, some of the AIS data must be "persistent"—it should exist beyond a single execution of a program. Furthermore, the persistent data of an application should be usable by more than one program; programs running concurrently should be able to read and write the data in a way that is safe and does not cause interference; and the data should also be recoverable in the event of a system failure.

Support for persistent data can be provided by the use of files and code implemented in the application programs. It is also possible that the file system component of an operating system could provide some of the desired functionality (for example, control of concurrent access). However, in most systems built over the last few years, the management of persistent data is performed by a database management system (DBMS). The most popular DBMS products at this time are relational database management systems (RDBMSs). However, as OOT becomes more popular, new database products are emerging which are either based on objects or are incorporating objects. These products are either object-oriented database management systems (OODBMSs), or object-relational database management systems (ORDBMSs). The latter are RDBMSs extended to incorporate ideas from OOT.

4.7.1 Object-Oriented Database Technology

At this time there is still no firm agreement on what constitutes an OODBMS. Relational systems are based upon a fairly simple mathematical model, and the evolution of these systems has been overseen by the founder of the model, E. F. Codd. Such a focal point may just be emerging in the OODBMS arena. R. G. G. Cattell is the author of one of the most respected books on OODBMS, *Object Data Management: Object-Oriented and Extended Relational Database Systems* [CATT94]. Cattell is also the chair of a committee, the Object Data Management Group (ODMG),[2] that is trying to establish a standard for what he calls "object data management" (ODM). ODMG has recently published *The Object Database Standard: ODMG-93* [CATT93]. The standard presents a framework for defining and querying objects, and for interfacing with OO programming languages. The committee is composed of representatives of many of the major commercial OODBMS vendors, and a number of these are committed to

[2] Not affiliated with the Object Management Group (OMG); however, the ODMG expects to submit its standard to OMG, the American National Standards Institute (ANSI), and other standards groups. OMG is responsible for the Object Request Broker (ORB), also known as Common Object Request Broker Architecture or CORBA. ORB is a communications specification to promote integration and sharing of objects, including persistent objects.

supporting ODMG-93 by early CY1995. Cattell's book on object data management [CATT94] contains a chapter that is an overview of the ODMG work.

An OODBMS, in order to claim the title, should have at least some of the following attributes.[3]

a. To be considered object oriented, it should provide the following:

    1. Abstract data types - the ability to define classes of objects that have associated "instance data" and operations (methods). The instance data of these objects may have a complex structure and may define relationships between objects.

    2. A class hierarchy and some form of inheritance.

    3. Communication among objects (messages).

    4. Object identification - an object should have a permanent identifier that does not change as long as the object exists, no matter how the state of the object changes.

b. To be considered a DBMS, it should provide the following:

    1. Transactions - support for consistent, conflict free, and durable updating and viewing of the object values. Durable updates mean that in the event of a system failure, it must be possible to restore the database to its most recent consistent state before the failure occurred.

    2. Persistence - the ability for objects to have a life span longer than the execution of an application program.

The DBMS requirements are minimal. To be truly useful, a DBMS should provide utility programs to support browsing and updating the database, a non-procedural query language, an optimizer so that non-procedural queries can be performed efficiently, an application program interface (API), as well as tools for database design, tuning, administration, and application generation and report writing.

There are now at least 10 commercial DBMS products on the market that can claim to be object oriented, based upon the attributes listed previously. In addition, a number of relational systems are planning on providing some aspects of object orientation, and some new products are being created that are called "object relational." SQL [ANSI92], the standard relational

---

[3] See [LOOM95] for a similar discussion presented from both a programmer and database perspective. Loomis's point is that object orientation blurs the lines between programs and databases.

data language, is evolving to support object orientation. SQL3 will provide abstract data types, and a later SQL standard is expected to define an even more object-oriented data language. RDBMS vendors can be expected to stay ahead of the standards effort if they perceive providing object orientation as important in order to stay competitive with OODBMS vendors.

4.7.2 Storage of Objects

In the best of all worlds, it would be possible for a programmer to implement a program that operates on objects without regard to whether they are persistent or not. However, since a great deal of the data manipulated in a program does not need to be persistent, this would place needless demands on an OODBMS. The next best possibility is for a programmer to be able to declare objects to be of a certain class and optionally to be persistent. These objects will be maintained by code that manages persistent data, but the programmer will be able to reference objects in a uniform way without regard to whether or not they are persistent. In this section we describe technical issues related to providing such a capability.

Generally speaking, users should not have to be aware of how data is physically stored in a database whether it is relational, object oriented, or some other model. Access to the data should be provided by an API. It should be possible for the DBMS vendor to change the way data is stored and, as long as the semantics of the API are preserved, users should not be affected. This is known as "physical data independence." In reality, (sophisticated) users are often required to be aware of some aspects of data storage in order to be able to better manage space utilization and performance.

OODBMS systems must maintain the instance data for each object. However, an object class is usually composed of instance data and methods, code that implements operations on the object. The OODBMS may or may not maintain the method "code" associated with an object. Some OODBMSs simply provide database services for managing the instance data. The method code is implemented in the application programs. In other OODBMSs, it is possible to define a persistent class that is composed of the instance data and methods. Maintaining the method code separately from the DBMS has the advantage that the methods are implemented in the same syntax as the application program, but the disadvantage that methods are separated from their associated data. That is, some aspects of an object are in one place and some in another (the database and the application program). In the case where the methods are managed by the DBMS, these advantages and disadvantages are reversed. This scheme, however, can also provide significant performance benefits since more complex processing can be performed by the database management system. This is especially valuable in a client-server environment

since it allows the application to off-load some processing on to the server and also cuts down on the amount of message traffic that must occur between the client and server.

The OODBMS may need to maintain additional information (for example, indexes) in order to support faster retrieval, scanning of a class of objects, and relationships[4] (IS-A, PART-OF, and user-defined relationships).

One of the major application areas perceived for OODBMSs is supporting applications involving complex objects, for example, computer-assisted design (CAD), computer-aided software engineering (CASE), scientific computing, and geographical information systems (GISs). In these applications, the instance data associated with an object can be quite complex (objects are frequently composed of other objects). There are basically two ways to store such an object, as a unit or in pieces with references connecting the pieces. If an object is stored as a unit, then a minimal number of disk accesses are necessary to retrieve it. The unit may be an image of a programming language object, or it may have a structure defined by the DBMS.

If an object is quite large, storing it as a unit might mean a number of database pages must be read even though only a small part of the object is being read or modified. If an object is in pieces, then a "lazy" approach can be used to access it. That is, a piece will be found and read only when that piece is referenced. The disadvantage of this scheme is that when the entire object is needed, it must be assembled. A considerable amount of processing could be necessary in order to support references to the object. OODBMS research is currently being directed at designing data structures and algorithms that will speed up this type of processing.

These two schemes (storing as units or storing in parts) can be simulated by an RDBMS, but this is an area where the RDBMS can yield poor performance. True OODBMSs try to store the parts of an object together so that a small number of disk accesses will be able to retrieve the entire object. In relational systems where data has been normalized, just the opposite effect is achieved. The components of an object are logically (and almost always physically) separated into tables so that in order to retrieve the entire object, it is necessary to perform a complex join or to use multiple queries.

An RDBMS can store an image of a programming language object using a BLOB (binary large object). A BLOB is a relational data type that is generally uninterpreted by the

---

[4] Loomis [LOOM95] refers to relationships such as IS-A and PART-OF as "containment" relationships. These should be stored directly by the OODBMS (as opposed to being symbolic and requiring a join as in relational systems).

RDBMS. That is, the RDBMS supports storage and retrieval of a BLOB but does not support other types of processing (queries, transformations, or updates except for replacement).

There are disadvantages to this image storage technique: the DBMS is tied tightly to a particular programming language, or is much less efficient when used by other programming languages. Since the OODBMS community seems to be focused on C++ and SmallTalk, this may not be perceived as a serious commercial problem. However, if the OODBMS community wants to be embraced by new types of users, they may need to address the problem of interfacing with a variety of programming languages. In particular, Ada users might prefer to use an OODBMS that is "language neutral" if they cannot get one that is Ada oriented. It is possible that the solution to language neutrality will be provided in the OMG's ORB context. This specification attempts to define a mapping between a general type structure and a particular implementation's representation. If OODBMS and RDBMS vendors participate in an ORB-based environment, then participating applications will not have to be concerned about a particular DBMS's representation of objects.

APPENDIX A.
OBJECT-ORIENTED ANALYSIS ACTIVITIES

A.1 REQUIREMENTS ANALYSIS

Requirements analysis has been described as “the process of determining what the customer wants a system to do” [HUTT94, p. 10]. When the customers and their wants are construed broadly enough, this provides as fair a definition of requirements analysis for OO software systems as for any other sort of systems engineering. Software requirements analysis need not differ much between object-oriented software systems and more traditional software systems since analysis of classes and objects, the distinctive features of OO systems, may be postponed until the object modeling stage. Even OO requirements analysis is predominantly functional in nature, consisting primarily of a determination of the essential functions that the system must perform, sometimes represented in terms of scenarios or “use cases” as described in Section A.1.3 on page A-3.

The attention given to requirements analysis, as a separate aspect of analysis, varies widely between different OO methodologies, from being assumed as background to being decomposed into detailed steps and operations. Some methodologies, such as Objectory, go into considerable detail on methods for eliciting systems requirements from users and other application domain authorities. Other methodologies, such as Wirfs-Brock’s, assume that a detailed requirements specification exists and base exploratory analysis on the extraction of information from that specification [HUTT94, p. 192]. Other methodologies, such as the Schlaer-Mellor method [HUTT94, pp. 165-176] and Coad-Yourdon’s Object-Oriented Analysis (OOA) [CDYD91], do not distinguish a separate requirements analysis process, beginning the analysis phase with domain analysis and object model building.

A brief list can identify most of the basic activities proposed by different OO methodologies for requirements analysis:

- Problem and requirements statement acquisition.
- Users and domain experts’ interviews.
- Use case model (or scenario) development.

- Interface development.
- Preliminary domain object model building.
- Prototype development for proof of concept.
- Customer sign-off on requirements models.

> *While most of these requirements analysis activities are optional, all are recommended. All the optional activities have proven effective in elaborating requirements and confirming them with the customer, although not all are needed in every program.*

Each of these activities is described briefly in the following sections. Only the first and last are required for every program. Obviously, some problem or requirements must exist for there to be any goal for a software development program. In addition, customer sign-off exists as a requirement of milestone review at the end of phases 0 and 1 of the AIS lifecycle in DoD Instruction 8120.2, insofar as the AIS milestone decision authority (MDA) is seen as a customer. Additional sign-off by end-user customers is an optional step that may further confirm the validity of requirements models. Some analysis of user input is also required as part of any requirements analysis performed under MIL-STD-498, though it does not mandate any specific form of input, allowing that

> This input may take the form of need statements, surveys, problem/change reports, feedback on prototypes, interviews, or other user input or feedback. [DOD94a, p. 14].

The optional activities may all contribute to satisfying this requirement, provided they involve the users.

A.1.1 Problem or Requirements Statement

Requirements analysis begins, in most OO methodologies, with some statement of the requirements. In its simplest form, this may be a statement of a problem to be solved by the system, with little or no information on what is required to solve that problem (e.g., "process payroll," "manage officer assignments," "manage materiel inventory"). Rambaugh's OMT begins from this minimal basis with the first step of "writing or obtaining the problem statement" [HUTT94, p. 156]. Given some statement of the problem or basic requirements, analysis at this stage proceeds to expand on it to establish the required functional behaviors of the system.

While most OO methodologies focus on *new* development, much of the current DoD AIS development involves *migration* of legacy systems, and the original written requirements are often no longer available for these systems. For some such projects, the best initial statement of requirements may simply describe the problem of reengineering the legacy system to reproduce its functionality within specified constraints, including modernized hardware, software, and engineering practices. In other cases, the requirements for a migration system may include new capabilities or merged capabilities from other systems in addition to modernization of a legacy system. Most basic OO requirements analysis processes will still be applicable to these projects, although they will benefit from supplementation with techniques and tools that are explicitly designed for reengineering, and possibly reverse engineering, of legacy software. This issue and others pertaining to software reengineering are treated in a companion report on reengineering systems strategies [IDA95d]. Here, we review only those requirements analysis activities that are of general applicability. When reverse engineering is required to help establish requirements, it may accompany or precede the ordinary requirements analysis activities.

A.1.2 User and Domain Expert Interviews

After some statement of the system problem or requirements is obtained, many methodologies prescribe some type of analysis of required system functionality. Such interviews may focus on eliciting or elaborating statements of needs, or they may engage the domain experts in more formal analysis activities such as "use case" analysis, user interface design, or prototype evaluation, as described in the next sections.

Interviews with users and other experts on the system's domain and intended operations are widely recommended as one means of determining and elaborating the system requirements.

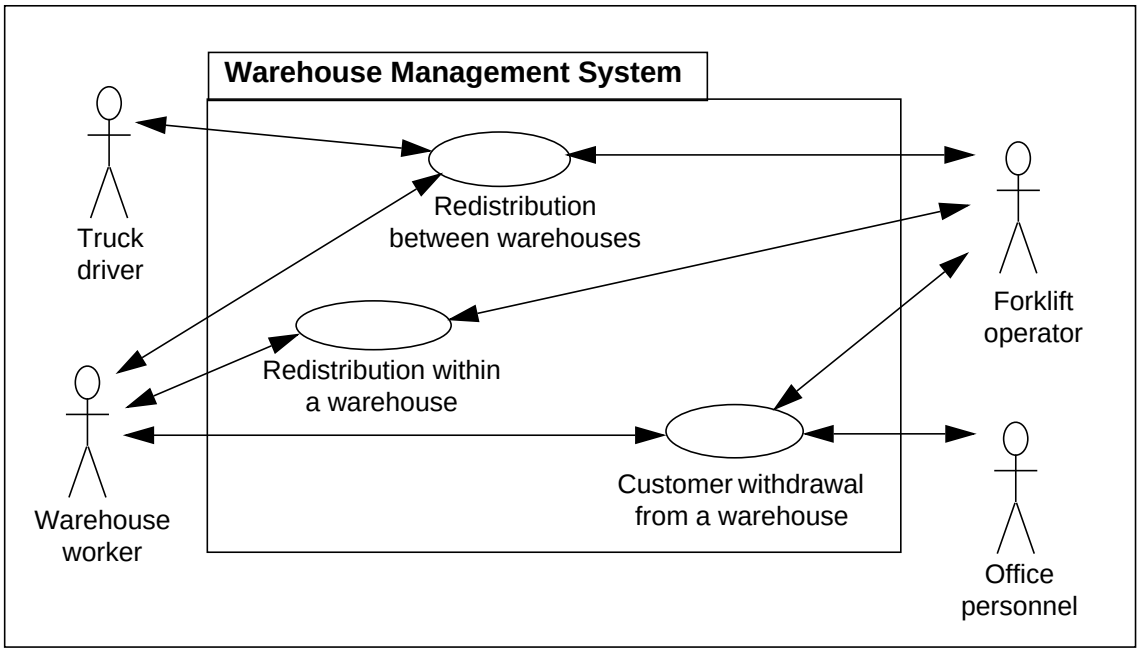
A.1.3 Use-Case Model Development

The development of "use case" scenarios in requirements analysis, as pioneered by Ivars Jacobson [JACO93], forms the foundation for all subsequent analysis and design in the Objectory method [HUTT94, pp. 107-117]. Recently, this technique has also been recommended by Grady Booch for both requirements analysis and for class object identification during domain analysis [BOO94a]. Rumbaugh's OMT advocates a very similar technique focusing on analysis of typical user dialogues called *scenarios* [RUMB91]. At this point, we focus on Jacobsen's use case analysis because it is the most detailed and

because he clearly places it in the requirements analysis phase, while Rumbaugh does not. The closely related activity of scenario analysis in OMT is discussed in Section A.3 on page A-40, the discussion on dynamic modeling, reflecting its place within the OMT methodology.

A use case is a behaviorally related sequence of transactions in which an external actor (a person or a machine) engages in a dialogue with the system [HUTT94, p. 109]. Use cases are intended to capture the functionality of the system as seen externally from the viewpoint of the users that will interact with it. The development of use cases is usually iterative and should involve the users or domain experts interacting with software developers. Subsequent domain analysis can then utilize the use case scenarios as sources of essential objects, classes, and their interactions.

Jacobson composes *use case models* from a set of actors and the use cases in which they participate. A diagramming technique is used for representing the actors (outside the system) and their relations to the use cases, as partially illustrated for his example of a warehouse management system in Figure A-6 [JACO93]. The system is represented by a labeled



Source: [Jaco93] **Figure A-1. Illustration of a Use Case Model**

box containing ellipses representing the use cases of the system, and surrounded by the actors external to the system connected by arcs to the use cases with which they are involved. Actors represent classes of individuals that are distinguished by the different roles

they play within the system. Actor roles are most commonly filled by persons, although other agents, such as external computer programs or system, may also play the role of actors. There is a many-to-many relation between actor classes and the individuals that may play their roles, i.e., a single individual can perform the roles of multiple actor classes, and many individuals and different kinds of individuals (e.g., human, software) can often take the same actor role.

A use case is specified by a description of its related flow of events in the order in which they occur. The different steps or events may be numbered (at multiple levels), domain object terms may be italicized, and alternative courses of events in the same use case may be separately identified at the end of the most common or default course. An example adapted from [JACO93] for the "customer withdrawal" use case of the warehouse management system example is shown in Figure A-2.

(1) The *office personnel* inserts a request for a customer withdrawal at a certain date and a certain warehouse.

(2) The information filled in by the *office personnel* is customer, delivery date, and delivery place. The *office personnel* person selects items from the browser and adds the quantity to be withdrawn from the warehouses. The browser can here show only the items of the current customer to the office personnel.

(3) The following criteria are checked instantly:
   a. The customer is registered,
   b. The number of items ordered exists in some warehouses,
   c. The customer has the right to withdraw the items.

(4) The system initiates a plan to have the items at the appropriate warehouse at the given date. If necessary, transport requests are issued. The items are reserved three days in advance.

(5) At the date of delivery a *warehouse worker* is notified of the withdrawal.

(6) The *warehouse worker* issues requests to the *forklift operator* to get the items to the loading platform. The *forklift operator* executes the transportation.

(7) When the customer has fetched his items, the *warehouse worker* marks the withdrawal as ready. The items are removed (decreased) from the system.

Alternative courses:

There are not enough items in the warehouses.
The office personnel is notified and the withdrawal cannot be executed.

The customer has no right to withdraw an item or the customer is not registered.
Notify the *office personnel*. The withdrawal cannot be executed.

Figure A-2. Customer Withdrawal Use Case

Commonalities may be abstracted from multiple use cases to create abstract use cases which can be referenced by those use cases sharing their activity sequences, thereby reducing duplication. But to avoid proliferation of use cases and keep the use case model simple, it is important to ensure that every use case, including abstracted ones, constitutes a complete logical sequence of events.

Use cases may be integrated with interface descriptions to better describe the actions taken by actors interacting with the system. These interface descriptions may be linguistic, graphical, or even implemented as a graphical user interface (GUI) prototype, as discussed in Section A.1.4 and Section A.1.6. Use cases may be further elaborated by a domain object model that begins to structure the objects referenced by the use cases, as discussed in Section A.1.5.

Use case (or scenario) analysis is increasingly recognized as a valuable tool for many aspects of development, including object modeling, dynamic modeling, and testing, as well as requirements analysis. It may be considered a best practice, though it is not incorporated in all OO methodologies.

A.1.4 Interface Description Development

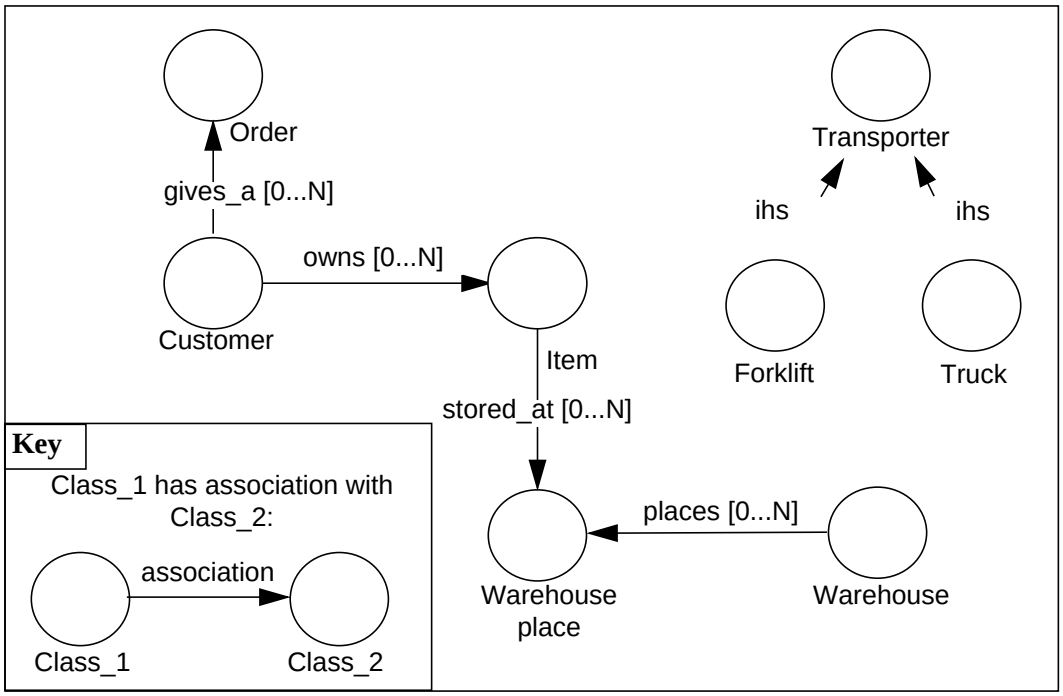
During interviews or analysis meetings with users and domain experts, the intended interactions between actors and the system in a use case may be clarified by use of user interface descriptions [JACO93]. These may be simple static sketches of screen displays, dynamic interface mock-ups, or actual front-ends to functioning prototypes. Jacobson emphasizes that such interface descriptions are best developed after the basic corresponding use case, to avoid forcing the use cases to fit some interface design preconceptions. Other system interfaces, such as communications protocols and standards, may also be described at this stage

Interface descriptions greatly facilitate effective communication on expected functionality between developers and customers. While highly recommended, they are not essential for every interface or every program.

A.1.5 Preliminary Domain Object Model

While most OO methodologies do not give any explicit attention to the domain model during requirements analysis, at least one approach (Objectory) recommends devel-

oping a “domain object model” during this phase. This domain object model is intended to lay the initial foundations for a logical view of the system in terms of application domain objects. It can help define the concepts used in the use cases or other scenarios, and it can provide the structure for any prototype that may be developed at this stage. A possible domain model in Jacobson’s notation for the warehouse management system example [JACO93] is illustrated in Figure A-3. The circles represent classes whose names appear on the labels beneath them and whose relations to other classes are indicated by labeled directed links between the class icons. The arcs, as usual in object models, represent associations between different types of objects. An annotation of “ihs” on an association arc indicates the inheritance association. One technique to aid building this sort of model, recommended in [JACO93], begins with free-form sketches by users and/or domain experts of their views of the system, followed by discussions designed to elicit the basic domain objects underlying those views.



Source: [JACO93]

Figure A-3. Preliminary Domain Model for a Warehouse System

In Jacobson’s Object-Oriented Software Engineering (OOSE) and in Objectory, this optional preliminary model is superseded in the next phase of development by a new object model, the *robustness model*. The preliminary domain object model is deliberately limited in scope just to provide any necessary clarification of the use cases. It is typically limited

to object names, attributes, and static associations, excluding object methods and responsibilities and dynamic associations. The intent is to avoid premature commitments to classes and to ensure that the final domain model is robust and benefits from a thorough analysis.

> *Building a preliminary domain model during requirements analysis is optional even in Jacobson's method. Its use should be determined by the needs and interests of the developers during this stage.*

A.1.6 Prototype Development

Prototypes developed during requirements analysis can serve a variety of purposes: training in OO concepts and development, clarification of an original vision of intended system functionality, concrete illustration of system concepts for eliciting user or customer feedback, proof of viability of certain system concepts, or even creation of a sales tool for promoting the system with potential customers or sponsors. Prototypes are, by nature, limited in scope, being incomplete implementations of selected aspects of the target application. They may focus on the user interface in order to support elicitation of user feedback on basic functionality from an operator's perspective, or they may focus on identified risk areas, such as specialized algorithms or processes, in order to test elements of the system that are perceived to carry substantial risk. Prototypes can serve these purposes at the same time that they are providing training in OO concepts and techniques for members of a development team that are transitioning to OOT from other software development paradigms.

Prototypes are intended to be quickly implemented, and the choice of a suitable prototyping environment may be crucial to this goal. The OO programming language Smalltalk, with its extensive object libraries and development environment, is often cited as an effective prototyping tool, even for projects whose target implementation language is different [TAYL92]. Especially effective for such rapid prototyping are the latest visual, parts-oriented environments built on top of Smalltalk, such as IBM's *Visual Age*, Digitalk's *Parts*, and Parc Place's *Visual Works*. Other languages or tools may prove comparably effective, provided they are adequately supported with object libraries, GUI tools, and an environment designed for rapid prototyping. Regardless of the language or environment used, it has been observed that an OO approach has advantages in rapid prototyping over function-based software development since the object classes and methods of an OO prototype may easily be a subset of the classes and methods of the full application, while a function-based

decomposition suitable for a prototype may well be inappropriate for the full application [RUMB91, p. 145].

Several methodologies identify the benefits of prototyping, and personnel in a wide variety of specific OO projects have expressed satisfaction with early rapid OO prototypes for pedagogical purposes as well as system analysis [TAY92]. Booch goes so far as to identify prototypes as the primary products of the requirements analysis (or conceptualization) phase [BOO94a, p. 250]. Regardless of the methodology employed, early prototyping has much to recommend it in OO systems development.

Prototyping is highly recommended throughout the OO development process, especially during requirements analysis. It is identified as an option for all program strategies during the entire lifecycle management process by DoD Instruction 8120.2.

A.1.7 Requirements Analysis Deliverables

Deliverables to be generated or updated by the end of each iteration of OO requirements analysis may include the following:

- Requirements specification
- Use case models or scenarios
- "Domain model" (An initial model of objects representing domain entities is an optional part of requirements model for some methods, e.g., Objectory.)
- Interface descriptions
- Prototypes

A requirements specification may include all of the other deliverables, or only a subset of them, depending upon which activities were undertaken during this phase.

When an information system is developed in accord with MIL-STD-498, the requirements deliverables must be provided in a form that includes all applicable items in the System/Subsystem Specification (SSS) Data Item Description (DID), DI-IPSC-81431 [DOD94a, DOD94b]. The requirements items for this specification are as follows:

3.1 Required states and modes

3.2 System capability requirements [including functions, subjects, and/or objects]

3.3 System external interface requirements

3.4 System internal interface requirements

3.5 System internal data requirements

3.6 Adaptation requirements

3.7 Safety requirements

3.8 Security and privacy requirements

3.9 System environment requirements

3.10 Computer resource requirements

3.10.1 Computer hardware requirements

3.10.2 Computer hardware resource utilization requirements

3.10.3 Computer software requirements

3.10.4 Computer communications requirements

3.11 System quality factors

3.12 Design and construction constraints

3.13 Personnel-related requirements

3.14 Training-related requirements

3.15 Logistics-related requirements

3.16 Other requirements

3.17 Packaging requirements

3.18 Precedence and criticality of requirements

The products of OO software requirements analysis fit under requirements item 3.2, System capability requirements.

Like other MIL-STD-498 requirements, this formal deliverable need not be completed in either a single build or prior to the traditionally subsequent activities of software design, implementation, and testing. Regarding required development activities, this standard allows the following:

> Many of the activities may be ongoing at one time: different software products may proceed at different paces; and activities specified in early subsections may depend on input from activities in later subsections [DOD94a].

Thus, it is quite compatible with the iterative style suited to OO development. More specifically, the SSS DID requirements analysis deliverable may not be complete until after multiple iterations of the development cycle. Parts of the SSS DID requirements will also depend on the input of other OO development stages. Many computer resource requirements, for example, are ordinarily not determined during OO development until design. OO analysis typically proceeds without concern for the details of the specific hardware and software environment, such as the operating systems or programming languages used.

A.2 OBJECT MODELING

A.2.1 General Features

Definitions

Object modeling focuses upon development of a model of the classes and objects in the application domain, their attributes, associations, hierarchies, responsibilities, and collaborators. “Object modeling” is both a generic term for these activities and the name of one stage of OO analysis in Rumbaugh’s OMT methodology. Other methodologies commonly use a different term to mean the same thing, as Objectory uses “robustness analysis” to emphasize its function of “finding a robust and extensible structure for the system as a basis for construction” [HUTT94, pp. 112-113]. In the Shlaer-Mellor method, object modeling is a part of “application domain analysis” and contrasted to other “service domain” analyses, where the activities of the latter are often associated with the design phase in other methodologies [HUTT94, pp. 166-170]. The Wirfs-Brock Responsibility-Driven Design method divides into two stages. The first, an “exploratory” stage (identifying classes, responsibilities, and collaborators), is included in what we call object modeling. The subsequent “analysis” stage includes some object modeling activity (establishing hierarchies), as well as other activities (establishing subsystems and protocols) that may better be included in a design phase [HUTT94, pp. 192-193]. Coad and Yourdon have little to say about requirements analysis and their whole analysis phase corresponds to what we are calling object modeling [CDYD91].

Diagram Notation

The models developed during object modeling are most commonly represented by diagrams, although they might also be described by natural language text or a data dictionary. In fact, object modeling tools typically support both diagrams and a corresponding data dictionary. Object models are static views of classes, objects, their attributes, responsibilities, associations, hierarchies, and (under some methods) collaborators. Classes and objects are represented by different types of annotated icons in the diagrams of different methodologies. Class icons from some of the more popular methodologies are illustrated in Figure A-4. Most diagrams of object models include the names of the attributes and the operations, methods, and services of a class within the class icon in order to identify its principal distinguishing features. Jacobson is an exception to this rule in drawing his attributes as links connected to the types of their arguments [JACO93], keeping the basic class icon fairly simple. The attribute and operation designations in other class icons, how-

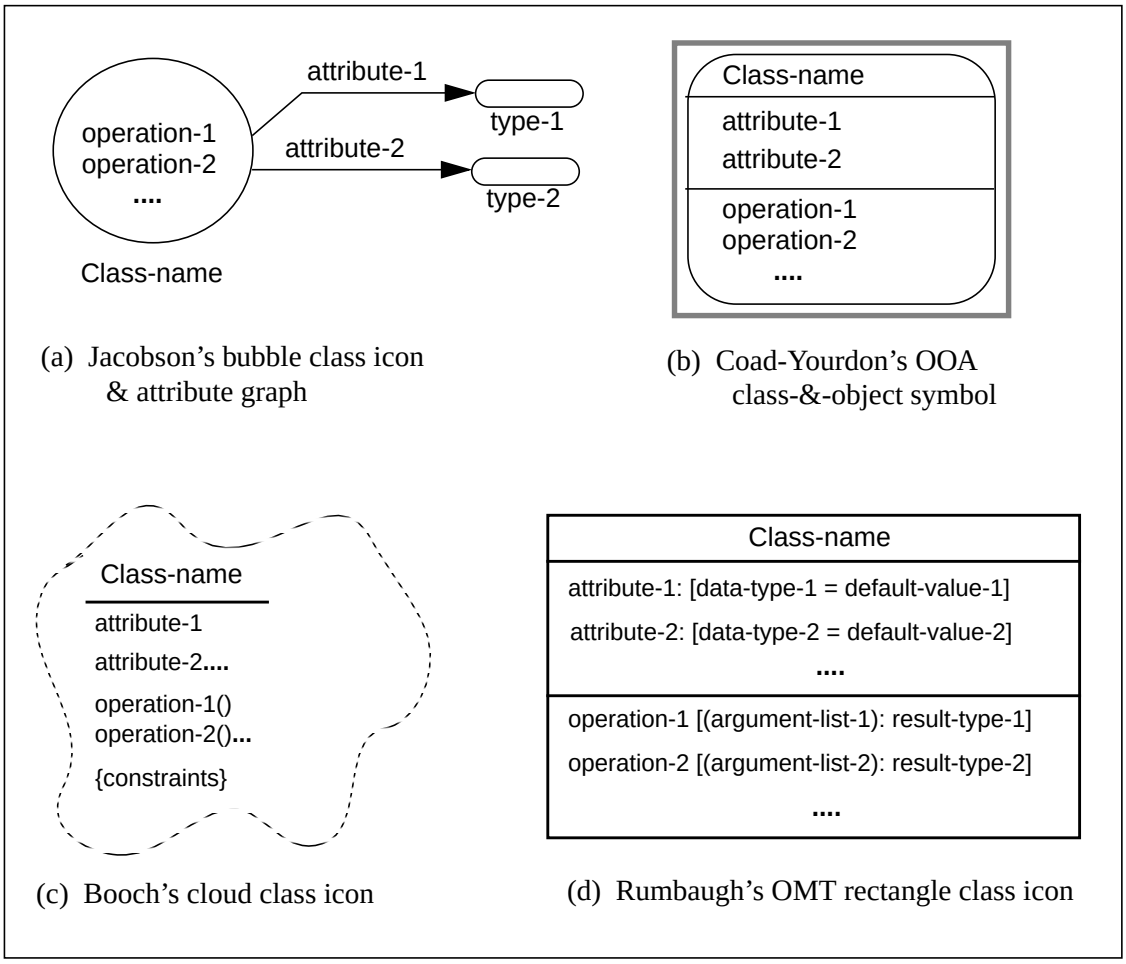
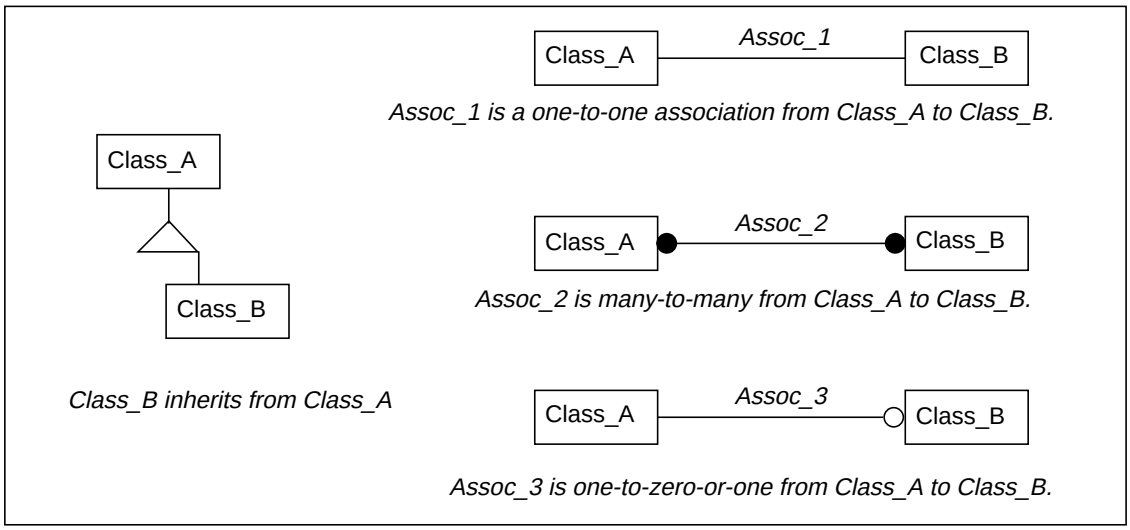


Figure A-4. Class Icon Examples

ever, are treated as optional, so that these class icons may be simplified when warranted. Individual object icons are usually minor variants on the corresponding class icon. Coad-Yourdon's diagrams also distinguish classes that may be instantiated directly, represented by the class-&-object symbol of Figure A-4, and those that are abstract classes, using a variant on this icon, without the outer box. Each of these object model diagramming techniques is most effectively utilized within a CASE tool that is designed to support it. Such tools are readily available commercially for all the most popular methodologies, though they vary widely in their features.

Diagrams of object models contain structures that correspond to association structures in the real-world domain that they model. The structures in object model diagrams are composed from class and object nodes and the arcs between them representing association, such as subclass inheritance, part-whole relations, and others. Figure A-5 illustrates diagramming conventions for some of the possible links between classes from one of the more

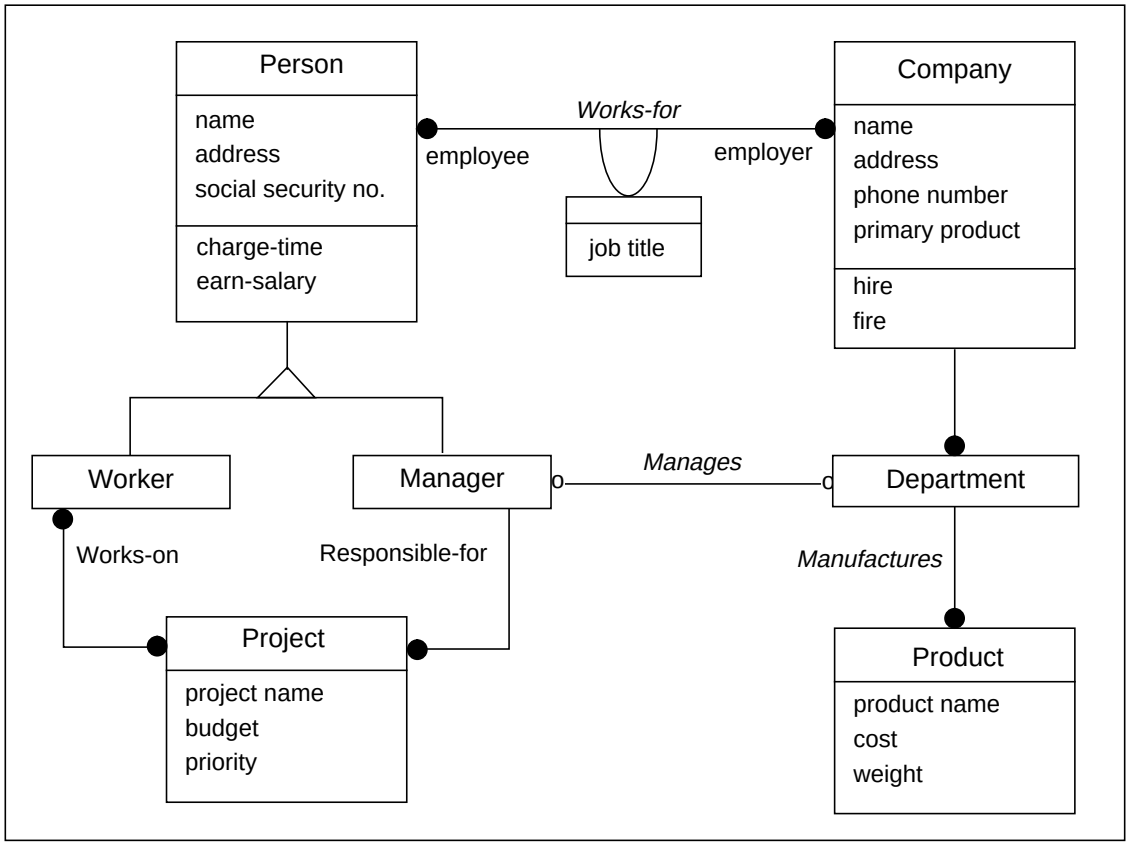
popular OO methodologies [RUMB91]. Considerable variation exists between methodologies on the precise conventions for annotating arcs to indicate their types and qualifications. In Rumbaugh's notation, cardinality of the associations can be indicated by the filled (many) and unfilled (zero-or-one) circles on the ends of their arcs as shown, or by numbers or ranges of numbers. More discussion of object/class associations follows in Sections A-2.5, A-2.6, and A-2.7 for the different association types of inheritance, part-whole relations, and other associations.



Source: [RUMB91]

Figure A-5. Examples of Class Link Diagram Conventions

The simplified example in Figure A-6 of an object model from [RUMB91] illustrates one approach to diagramming object structures applied to a business domain. It illustrates a many-to-many association *works-for* between the *Person* class and the *Company* class, a one-to-many association between the *Company* class and the *Department* class, and a zero-or-one to zero-or-one association of *Manages* between the *Manager* class and the *Department* class, among others. It illustrates inheritance from the *Person* class by the subclasses, *worker* and *manager.* Classes are represented at varying levels of detail: solely by class name, with attributes, and with both attributes and operations.



Source: [RUMB91]

Figure A-6. Example Object Model with Inheritance and Associations

Activities

The principal activities and techniques used in building such object models are all just means of identifying their components and putting them together into a coherent picture, as can be summarized by the following list:

- Identify classes and objects.
- Identify responsibilities (services or operations) associated with classes.
- Identify attributes, the data that describe objects.
- Identify inheritance relations, the generalization and specialization of classes.
- Identify whole-part structures or aggregations of classes/objects.
- Identify other associations.
- Identify “collaborations” between objects/classes.

- Maintain a data dictionary.
- Perform CRC card analysis (class, responsibility, collaborator identification).
- Identify constraints and rules.
- Partition the analysis model (e.g., interface, entity, control).

The first six activities are essential in building an object model of the application domain; they identify the basic elements of an object model: classes, operations, attributes, inheritance, aggregation, and other associations. The order in which these activities occur and their prominence during the analysis stage vary between different methodologies. "Data-driven" methods (e.g., Coad-Yourdon's OOA and Rumbaugh's OMT) focus more on the attributes and associations of classes, while "responsibility-driven" methods (e.g., Wirfs-Brock) focus on responsibilities of classes and collaborations between them.

Identification of collaborations—the use of the services and operations of one class by another—is not included as part of the object model by all methods. Booch's method includes collaboration identification as his analysis stage and incorporates collaboration links on the "class diagrams" that represent his object model [BOO94a]. OMT, however, has no place for collaborations on its object model diagrams, relegating them solely to the event flow and event trace diagrams used during its dynamic modeling stage. Thus, modeling collaborations may be considered an optional part of the object modeling stage, although it is essential somewhere in the development process for every mature OO development methodology.

Maintenance of a data dictionary occurs concomitantly with the other activities in any good development environment since object model components should be defined as they are identified and maintained in a common place for convenient reference. CRC card analysis is one widely adopted technique for structuring some of these basic activities: identifying classes, their responsibilities, and their collaborations with other classes. The constraints and rules identified by the next activity in this list are rather specialized features of objects and classes that are not addressed by every methodology and may not appear in every object model. The last activity listed, "Partition the analysis model," addresses partitioning the whole object model into subareas, which is approached in different ways by different methodologies. Some methodologies prescribe an initial partition of the whole object model prior to its construction (e.g., Jacobsen [JACO93] and Shlaer-Mellor [HUTT94]), while others reserve the explicit groupings of objects until the last step of analysis (e.g., [RUMB91, pp. 168-169]).

Development of an object model is an universal feature of OO analysis. While classes and some of their aspects (attributes, services, associations, inheritance, aggregations, and collaborations) are identified at this stage by every OO methodology, considerable variation exists on the emphasis given to different aspects.

A.2.2 Identifying Classes and Objects

A wide variety of techniques have been proposed to aid in identifying suitable objects and classes for inclusion in the object model and eventual implementation in the target application. Some methodologies recommend beginning with the text of the problem or requirements statement and extracting noun phrases as candidate class names (e.g., Wirfs-Brock [HUTT94, p. 192], Coad-Yourdon [CDYD91, p. 60], OMT [RUMB91, p. 153]). Others caution against applying this technique automatically since natural language text can be misleading, and a more thorough analysis is usually required to confirm the suitability of such candidates [BOO94a, p. 159-160]. Several other sources are commonly recommended as places to look for relevant objects and classes, including interviews with domain experts, examination of similar existing applications, first-hand observation of domain practices, and literature research in the application domain, and, in the case of a legacy system, any existing documentation such as entity-relationship diagrams that may be available.

Use cases, or scenarios, provide another major source of candidate objects and classes. Jacobson's OOSE methodology advocates identifying *all* of the required domain entity objects from the use cases developed during requirements analysis [JACO93, p. 184]. Booch uses scenario analysis in conjunction with general domain analysis to identify classes and objects. His domain analysis identifies classes and objects for the general problem domain, whereas scenario analysis focuses upon scenarios of the specific target application in that domain [BOO94a, pp. 253-254]. Thus, scenario analysis may pick up classes/objects overlooked by domain analysis, as well as weed out domain classes/objects that are not used in the target application. However, at least one methodology (i.e., OMT) neglects these object modeling benefits of scenario analysis in favor of using scenarios solely to extract dynamic behavior. The OMT methodology presented in [RUMB91] fails to suggest using scenario analysis for identifying classes/objects, although its iterative approach implies the potential of feedback from scenario analysis in dynamic modeling to object modeling. We see a full range of involvement of scenario modeling in object/class identification among different OO development methodologies, from being the only approach to being ignored.

Scenario modeling and its potential benefits are discussed further in the section on dynamic modeling, Section A.3 on page A-40.

Some methodologies identify particular types of classes/objects to look for, whether in requirements statements, scenarios, domain literature, or interviews. Coad-Yourdon's OOA suggests looking for part-whole and subclass structures, interactions with external systems, devices, events and things that must be recorded, roles played by agents, locations or sites, and organizational units [CDYD91]. The Shlaer-Mellor method has been described as emphasizing tangible things, roles, events, and interactions as the primary sources of candidate classes/objects [BOO94a, p. 155]. OMT cites two general types of classes: physical entities such as houses, people, and machines; and concepts such as trajectories, seating assignments, and payment schedules [RUMB91, p. 153]. Many such sets of class categories may be useful for suggesting candidate classes for inclusion in an object model.

After identifying object/class candidates, most methodologies include a winnowing process in order to narrow down the candidates to those object and classes that are truly required for the specific application. Specific reasons that have been identified for excluding or restructuring candidate classes include the following [BOO94a, CDYD91, RUMB91]:

- Redundancy appears in multiple candidate classes.
- Domain class is irrelevant to target application.
- Class is imprecise or vague.
- Class name better describes an object attribute, operation, or implementation construct.
- Class name describes a role of an entity, better modeled as an association.
- Attributes of candidate class do not apply to all class members.
- No attributes of candidate class are essential to the application.
- Candidate class with only one attribute may be better modeled as attribute.
- Operations of candidate class do not apply to all class members.

The application of these criteria should continue throughout the analysis process and its iterations as new objects, classes, attributes, associations, and responsibilities are identified and refined.

A.2.3 Identifying Responsibilities, Services, and Operations

Definitions

Responsibilities, services, operations, methods, class member functions—all these terms refer to essentially the same things: the behavioral aspects of classes and objects, in contrast to their declarative aspects of having certain attributes or associations. Services have been described as providing the "processing" component of an OO data processing system, where attributes provide the data [CDYD91]. The terms "responsibilities" and "services" are used in OO analysis and design to emphasize the implicit contractual relations between classes to provide the specified functionality. From a client-server perspective, the operations of an object/class provide services for other objects/classes that play the role of clients using those services. "Operations" is a fairly neutral term referring to object/class procedural capabilities more from the perspective of the object/class itself, irrespective of its use elsewhere. "Methods" and "class member functions" are largely language-specific terms for operations from the perspective of implementation, referring to specific algorithms or code that implement class operations. Class operations implemented in the OO language Smalltalk are referred to as "methods," while those in the C++ language are referred to as "class member functions" [LIPP91]. In Ada 95, they are simply called "operations," although these are divided into the separate kinds of procedures and functions [INTR93].

Types of Operations

Operations come in a wide variety of types. Some of the more commonly distinguished categories of operations include the following:

- Constructors and destructors
- Attribute access (information request)
- Association link traversal
- Calculations
- Printer output
- Screen updates
- Action requests
- Event monitoring

- Event notifications

Constructors and destructors create and destroy individual objects of a class by allocating and deallocating memory, instantiating data structures, and setting links in accord with the class definition. Methodologists commonly advise to use uniform methods for constructors and destructors across all classes in order to reduce both risks and development costs. Attribute access methods must be provided for all encapsulated attributes to provide protected means of reading, writing, and updating their values. However, many obvious operations, such as these attribute accesses and association accesses, may be neglected in the analysis phase, though they must be fully accounted for during design and implementation. Some attributes, such as the public attributes in a C++ implementation, do not need special access operations since they are directly accessible through the built-in language naming conventions. However, use of publicly accessible attributes is a violation of the OO convention of data encapsulation, and is generally discouraged in OO programming.

Traversing association links, such as the links *Works-for*, *Manages*, and *Works-on* in the example of Figure A-6 on page A-15, may be accomplished using operations for each association of a class. Ordinarily, such association-traversal operations are not separately identified in an object model, being implicit in the association links. Calculations of all types, from simple sums to complex mathematical algorithms applied to large arrays of scientific data, constitute one class of operations that are commonly included on object diagrams.

While many class operations function solely within the confines of the software, others interact with the external world. Printer output, display updates, graphical object controllers, and machinery controllers are just some of the many kinds of class operations that can directly affect the external world. Operations such as action requests, event monitoring, and event notification are other types of actions that may involve real-world interaction when involved in systems that monitor and/or control external events or processes. Internal information system activities may also fit these categories. Some changes to information, for example, may be modeled as events or as the results of actions by a user or an internal program object.

Diagram Notation

Operation names often appear directly below the attribute names in the class icons of the class/object diagrams of OO methodologies, as illustrated in Figures A-4 and A-6. The arguments taken by an operation may also be included on a class diagram or in the cor-

responding data dictionary. Certain types of operations, such as constructors, destructors, and attribute access operations, are commonly not represented on object/class diagrams since some services satisfying these needs may be assumed to exist for all classes and attributes. Listing of other operations on class diagrams is optional, depending on their significance to analysis and design. Regardless of whether they appear on diagrams, all operations should be listed in the object specifications or data dictionary.

Identification Techniques

There are many techniques for identifying class object operations. In fact, many other activities during the analysis phase support the identification of operations. Use cases or scenario descriptions can be directly analyzed to extract the operations that are often implicit in action verbs such as *calculate*, *determine*, *select*, *check*, *issue*, *notify*, *mark*. The classes and objects already identified may be considered as sources of activities characteristic of their types in the application domain. When operations are so derived from general domain knowledge of the activities of some class of objects, they must, of course, be tested for specific relevance to the targeted application. Some general activities commonly associated with a certain class of objects may not be relevant to the information system being developed. Many of an application's essential operations may be best identified during the detailed dynamic analysis, which models the target system's required behaviors, as discussed in Section A.3. But an initial set of operations suggested by general domain knowledge may be helpful in formulating such dynamic models. More specifically, the attributes of the object model may be a fruitful source of suggestions, based on knowledge of operations that are commonly performed upon them. Basic access operations may, of course, be assumed for all attributes. Candidate operations may also be refined by recognition of commonalities among already identified operations, which suggests a more general operation that might better be placed within a common superclass.

While basic object modeling of classes and attributes may be suggestive of many candidate operations, most of the nontrivial operations will have to be validated by the needs identified in dynamic modeling. Depending upon the methodology adopted, this dynamic modeling may occur prior to, concurrent with, or subsequent to detailed object modeling. In the Responsibility-Driven Design method of Wirfs-Brock, the provisions of operations and attribute values are both types of class responsibilities whose determinations are practically concurrent using CRC card analysis [HUTT94]. When object modeling precedes dynamic modeling, as in Rumbaugh's OMT, it makes sense to delay the selection of class operations until the end of the analysis phase, as OMT does. If detailed dynamic anal-

ysis is delayed until the design stage, as in Jacobson's OOSE, then selection of operations may be delayed until then, as OOSE prescribes [JACO93, p. 189]. Other methodologies offer a more integrated approach to object and dynamic modeling during analysis, as does the Booch method in its broad category of "identifying the semantics of classes and objects," which is part of its analysis stage [BOO94a].

The identification of operations during dynamic modeling is briefly summarized here, with detailed explication below in the separate subsection devoted to it. The interaction diagrams developed under many OO methodologies during dynamic analysis include events (requests for services) sent between objects. Other methodologies that lack explicit interaction diagrams may begin by listing required collaborations between classes, as in the CRC card analysis central to the Wirfs-Brock method. A collaboration graph generated from the refined CRC card analysis may also aid determination of appropriate operations to meet class responsibilities in these collaborations. Each event or collaboration can identify a candidate operation to satisfy its needs, though many such candidates may be redundant or trivial, so that they need not all be added to the object/class diagram. Some collaborations may only involve simple attribute access or update operations, which are already understood to exist. Responses may simply return values from previous messages or function calls which do not require any independent operations.

The other main product of most OO dynamic modeling—state transition diagrams—provides another source of candidate class operations. The actions or events on transitions between the states of a object are commonly modeled as operations and may be added to the object model when not trivial (such as attribute access operations) or already documented. Given the many sources of candidate operations, high levels of redundancy can be expected so that checking for the equivalence and subsumption of candidate operations may be a major part of operations selection. Identification of overlapping functionality in operations from different classes may also indicate a need for operation consolidation, such as placement of a comprehensive operation in a common superclass. Such inheritance of common functionality is one of the principal means of simplifying OO development, as further discussed in Section A.2.5 on page A-24.

A.2.4 Identifying Attributes

Definitions

Attributes of classes specify a type of property possessed by each object in that class. Common attributes of physical objects, for example, include color, weight, height,

width, and location. An actual instance of such a property possessed by a specific object is specified by assigning a value to the attribute of the object, e.g., the value “red” for the color of a particular vehicle. In OO systems, the data about objects are contained in their attribute values whose complexity can range from primitive numeric and alphanumeric values to complex composite structures such as images and video sequences. The attributes actually used for an object class in a particular information system depend on the nature of the object class and the requirements of the specific application.

When attributes qualify associations between two objects they are called *link attributes*. Rumbaugh gives the example of an association between *Stockholder* and *Company* having a link attribute of *number-of-shares* [RUMB91, p. 162]. Such link attributes are essential in specialized domains that use attributed relational graphs, such as certain approaches to image processing. But attributed links are just one of many possible refinements of class associations not covered by many methodologies and which are often neglected in practice.

Diagram Notation

Class attribute names appear directly below the class names in the class icons of the class object diagrams of most methodologies, as illustrated in Figures A-4 and A-6. The data type of the values (e.g., integer, string) taken by an attribute may also be included on a class diagram or relegated to a corresponding data dictionary. Individual object icons, in contrast to class icons, frequently display only attribute values, not attribute names or datatypes, as they are used, for example, in [BOO94a] and [RUMB91].

Some methodologies also support identification of default values for class attributes. A default value for a class attribute is one which may be assumed to be the correct value for that attribute for all objects in the class which do not have an overriding value specified. Such default values may be specified with the attribute on a class icon in the notations of Rumbaugh [RUMB91] or Booch [BOO94a]. While some OO programming languages do not directly support the specification of default values for class attributes, this capability can usually be programmed by encapsulating the attribute values and providing a default through an access function which is included in the operations of the class.

Access to attribute values for examination or updates is ordinarily provided only through explicit services, methods, or operations of an object/class in order to support encapsulation of the actual structure and format of attribute values. Such encapsulation is one of the basic features of OO systems, increasing maintainability by keeping changes to

underlying data structures isolated from other parts of an OO system. The actual attribute access services, such as *read* or *update*, are not ordinarily specified on object/class diagrams since it is understood that some such services exist for all class attributes.

Identification Techniques

Attributes of a class can be identified by inquiring about the general description of class members, which may be obtained from requirements specifications, general domain documentation, or interviews with domain experts. Rumbaugh notes that the attributes in such descriptions can often be identified by nouns followed by possessive phrases, such as "the color of the car." The different states that an object may pass through can also serve to identify attributes required to differentiate those states. A major source of class attributes is to be found in the data required of objects in order to execute their responsibilities. Thus, much of the determination of attributes will have to follow some analysis of class responsibilities, such as determined during scenario or use case analysis. Both Booch and Rumbaugh caution against excessive commitments to class attributes during the analysis phase, emphasizing the inclusion of only those attributes that are needed for identified operations in the application [BOO94a, RUMB91].

A.2.5 Identifying Inheritance Relations

Definitions

Inheritance relations exist between two classes when one is a subclass of another, that is, when all the objects in one class are also in the other class. The subclass is then said to *inherit* the attributes and operations of the superclass. Such inheritance relations are also commonly referred to as generalization-specialization relations: the subclass is a specialization of the superclass and the superclass is a generalization of the subclasses. The classes *Worker* and *Manager* shown in the object model of Figure A-6 on page A-15, for example, are both specializations of *Person*, inheriting its attributes, such as *name* and *address*, and its operations, such as *earn-salary.*

Inheritance relations can create class hierarchies of varying depth, breadth, and complexity. Single inheritance, where classes are allowed at most one generalization or ancestor, restricts the resulting hierarchies to tree structures,[1] as illustrated in Figure A-7. Multiple inheritance, which allows a class to have multiple generalization ancestors, supports more complex "tangled hierarchies." Multiple inheritance is directly supported by the

[1] Assuming that mutual inheritance and looping are excluded.

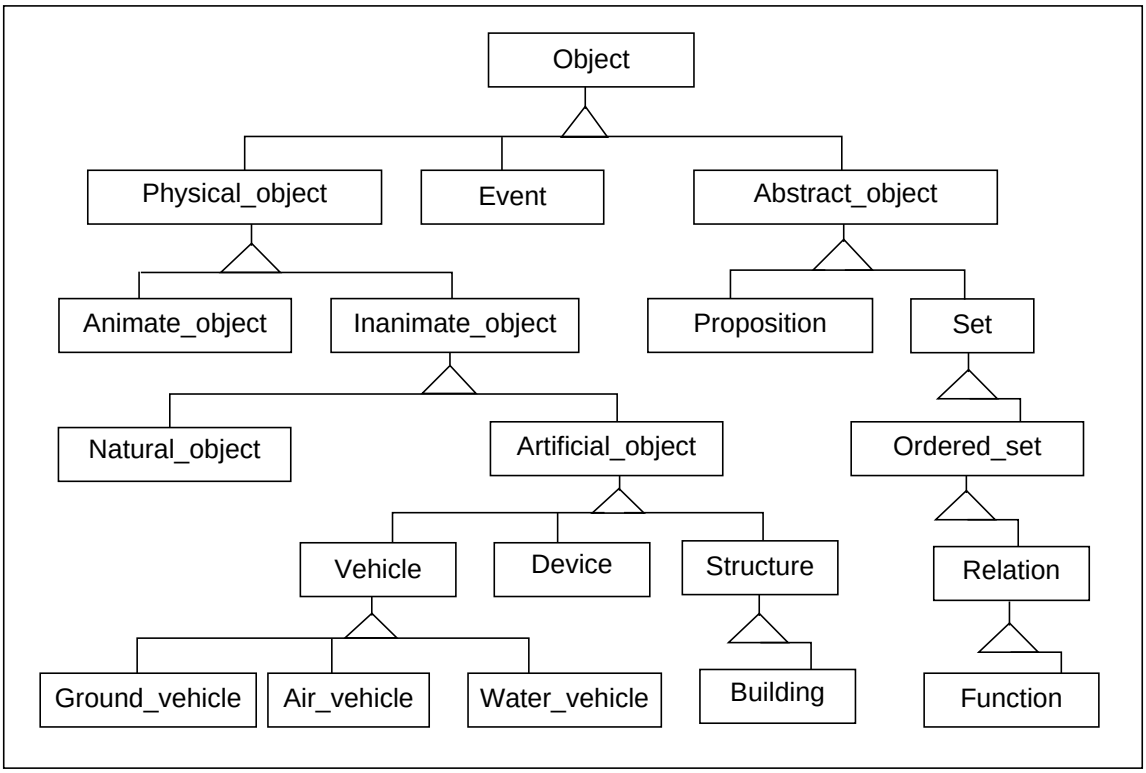


Figure A-7. A Single Inheritance Tree

OO languages C++, CLOS, and Eiffel, but is unsupported by Smalltalk and Ada. Ada 95 does not directly support multiple inheritance either, although some of the effects of multiple inheritance may be achieved by the use of the Ada *with* and *use* clauses to provide access to the operations and data structures of classes outside of the current class and its ancestors. Other workarounds for handling multiple inheritance when the implementation language does not support it are discussed by Rumbaugh [RUMB91].

Diagram Notation

Figure A-7 shows part of the upper levels of a single inheritance hierarchy. While this example is a *single* tree structure, having the common root *Object* for all classes, this is not a general requirement. Object models may also be composed of multiple hierarchies. Although some OO languages, such as Smalltalk, ultimately require a common root, this need not constrain the object model in the analysis stage since a common root may be added in the design phase if needed by the implementation environment. Multiple inheritance is a separate issue, as illustrated by Figure A-8, where the classes of *Amphibious_car* and *Sea_plane* each have multiple ancestors from which they inherit their dual capabilities, although all share the common ancestor *Vehicle*.

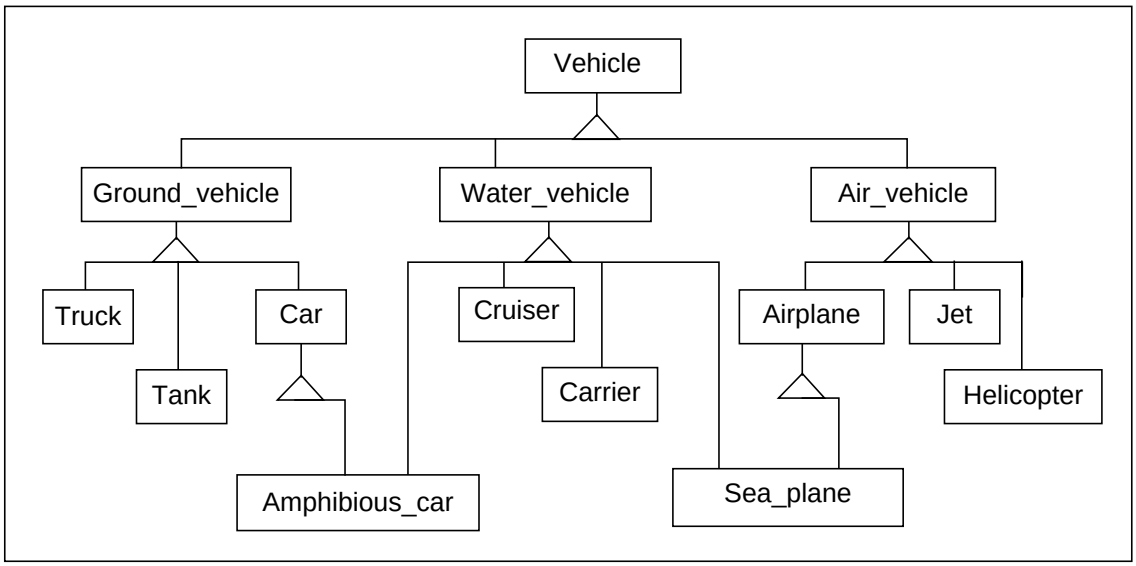


Figure A-8. A Multiple Inheritance Hierarchy

Purposes

Inheritance structures serve several purposes within OO systems: the organization of classes, reduction of redundancy, enforcement of consistency, and support of reuse. The generalization-specialization relations of inheritance structures provide a natural organization for the many objects of a typical OO application, aiding comprehension and simplification. Placement of common attributes, operations, and associations in the most general classes reduces redundancy in their specification and implementation. It also enforces a consistent representation and implementation for features common to multiple classes, ensuring functional compatibility between different classes with respect to those features.

Finally, the benefits of inheritance capabilities extend beyond any single project to support effective reuse of class components by multiple projects. Classes from class libraries or reuse repositories that offer potential for reuse often require some customization before they can fully meet the needs of a current project. Inheritance provides the capabilities for specializing an existing class by specifying a subclass that inherits desired features, overrides undesired features, and adds missing features. These basic inheritance capabilities create a very accommodating environment for adapting repository components to the needs of a current project, as well supporting reuse within a single project.

Identification Techniques

Inheritance relations between classes are often readily apparent from common sense or domain knowledge. However, many different inheritance structures are ordinarily compatible with this knowledge for any given domain. The best structure for an inheritance hierarchy depends on the point of view of the application and its context. Effective structuring of inheritance hierarchies is ordinarily aided by the analysis of commonality among the attributes and operations of candidate classes. An overlap among the attributes or operations of distinct classes may indicate the presence of a common superclass from which they can inherit these features. For example, imagine a model containing separate classes for ground vehicles, air vehicles, and water vehicles, all of which have common attributes for maximum velocity, maximum range, fuel consumption, maximum cargo load mass, and number of crew members, along with common operations such as range as a function of fuel amount and velocity. Then, if there is no common superclass, it makes sense to extend the model with one to cover all vehicles that includes all of their common attributes and operations.

If the attributes or operations of a candidate class fail to apply to all its members, this can indicate a need for a new subclass or subclasses to which the offending attributes or operations may be relegated. For example, if the class of tanks has attributes for turret dimensions but not all tanks of interest have turrets, then separate subclasses of turreted and un-turreted tanks may be indicated. Unnecessary classes may also be eliminated while structuring the inheritance hierarchy. Inessential subclasses may be recognized if all of their required structure and functionality can be expressed in a superclass. Inessential superclasses should be eliminated if they do not offer any common structure or functionality to their subclasses. This helps keep the hierarchy depth at a level suitable for the application.

Some guidelines that can help in constructing effective inheritance hierarchies are as follows:

- Use information about common services and attributes of classes.
- Never use inheritance overrides in a subclass to remove a superclass service.
- Do not violate the commonly accepted categories of the application domain.

Although these are all common sense guidelines, they still can be helpful to the novice and can influence the order of activities in object modeling.

Like other aspects of OO models, inheritance hierarchies are constructed iteratively.

As deeper analysis and design uncover more classes or required functionality, or reveal more commonality of structure and functionality, the inheritance structures may be changed to accommodate them more effectively. Because effective structuring of inheritance hierarchies requires considerations of commonalities of structure and function, it can be more effective to delay full-fledged inheritance analysis until after some analysis of both attributes and operations, as suggested by the first guideline ("Use information about common services and attributes of classes"). Some methodologies implicitly recognize this constraint, as Booch does by placing the identification of class relations, including inheritance, after identification of class semantics (structure and behavior) [BOO94a]. Other methodologies, such as Rumbaugh's OMT [RUMB91], prescribe an initial analysis of inheritance before identifying operations, presumably with the expectation of adjustment during iteration. Iteration naturally makes the development process quite forgiving and allows flexibility in the order of activities since the next iteration can adjust for changes due to activities of a prior iteration. Convergence towards a stable model is expected as iteration proceeds, although indecisive analysts risk what Booch calls "analysis paralysis," where they are stuck iterating their analysis endlessly [BOO94a, p. 255]. The cure for analysis paralysis, according to Booch, is to move on to prototyping, design, and implementation, and not attempt to get the analysis model perfect.

A.2.6 Identifying Part-Whole Relations

Definitions

The part-whole relation is a ubiquitous type of association between two objects in which one is a part of the other. It is most commonly exemplified in relations of physical containment or composition between physical objects, such as the pieces of a jigsaw puzzle or the components of an engine. Yet other types of part-whole relations find representation in ordinary discourse as well as in OO software. Social organizations ordinarily decompose into parts, such as the divisions, departments, groups, and committees of business organizations, and the divisions, battalions, platoons, or squadrons of different military organizations. The individual members of any organization may also be treated as parts of it, though there is some disagreement on the best representation of such relations in OO systems.[2] Less tangible entities, such as events, processes, propositions, and mathematical functions, may also participate in part-whole relations: complex events may decompose into subev-

[2] Rumbaugh argues against treating employees, for example, as parts of a company [RUMB91, p. 58], preferring a *Works-for* relation; Coad and Yourdon treat clerks as parts of organizations [CDYD91].

ents; processes decompose to subprocesses; compound propositions may decompose into primitive propositions; and a piecewise continuous mathematical function can be decomposed into continuous segments.

While any particular part-whole relation associates two individual objects, a generic part-whole association between two classes may be specified in an OO object model to indicate that each (normal) instance of one class has a part which is a member of the other class. We say that computers have CPUs, cars have wheels, lamps have bulbs, airplanes have wings, and ships have hulls—all to indicate that each object in the first class has one or more objects in the second as a part. Most specifications of part-whole relations for OO systems associate pairs of classes, which are then instantiated to specific part-whole instances as instances of those classes are created.

Part-whole associations have been labeled in different ways by different authors. Jacobson calls them *consists-of* associations [JACO93, p. 181]; Rumbaugh notes that they are often called *a-part-of* relations [RUMB91, p. 59]; and Booch refers to them as *has-a* relations [BOO94a, p. 180]. Structures composed of a whole, its parts, and their relations are commonly referred to as *aggregates*, and the grouping of parts to form a whole as an *aggregation*.

Diagram Notation

Part-whole relations between classes are represented in the object models of OO methodologies by special links between them, as illustrated for Rumbaugh's OMT in Figure A-9. Other methodologies use different elaborations on the links, expressing part-whole relations, though they often have a distinctive symbol (such as OMT's diamond ◊) to identify the whole or aggregate end of the whole-part relation. Part-whole relations can form large strict hierarchies or more complex looping and recursive structures. Large social institutions, such as government departments and big corporations, characteristically decompose into broad, deep hierarchies of part-whole relations. Recursive structures are more likely to appear in more formal systems, such as programming languages or window systems, in which one statement or window may have another as a part. An example of an object model including both recursive and non-recursive part-whole relations is presented in Figure A-10.

Identification Techniques

As with inheritance relations, part-whole relations are often readily apparent from

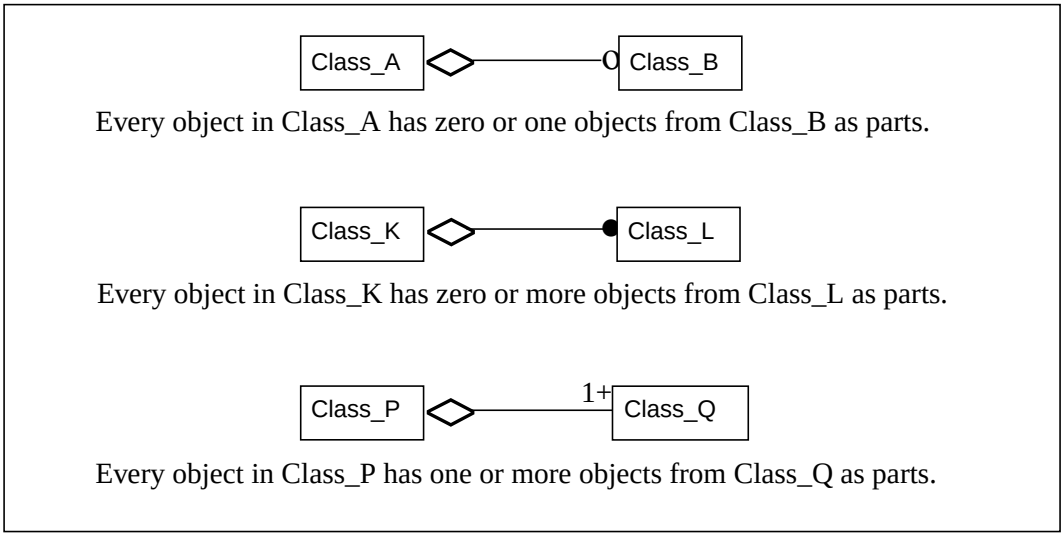


Figure A-9. Part-Whole Relations in OMT Object Models

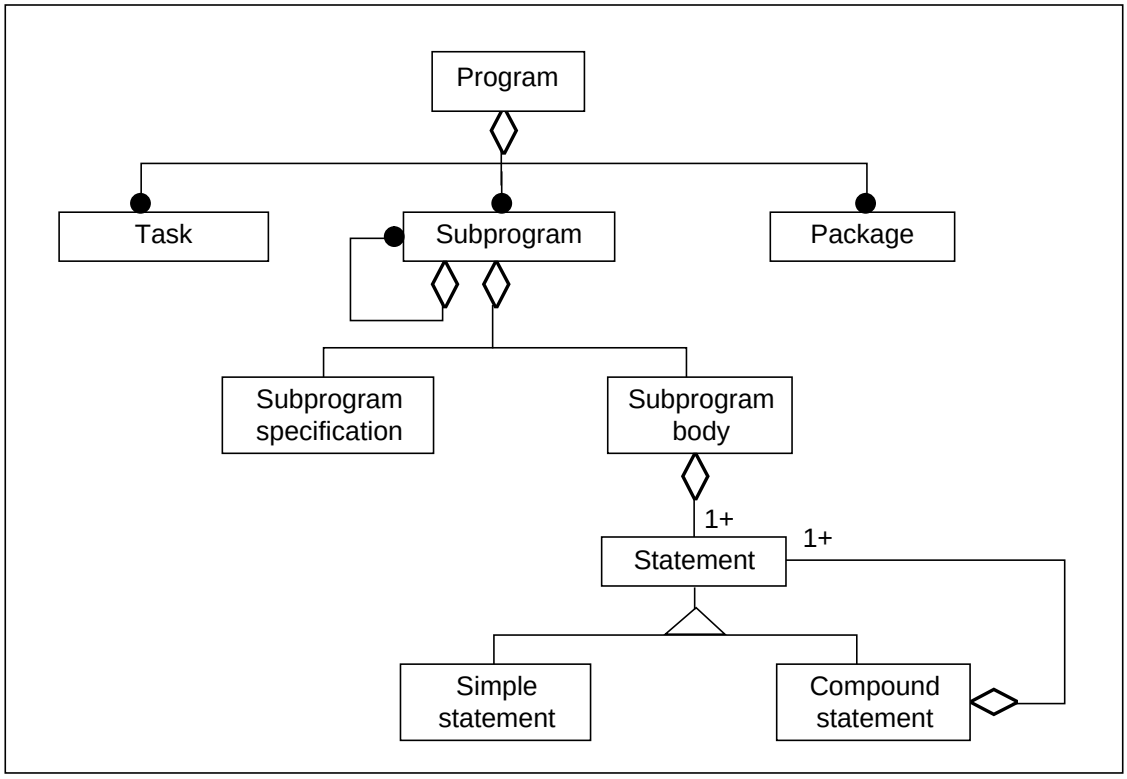


Figure A-10. Example of Part-Whole Structures

common sense or domain knowledge. Domain experts and any available OO analysis results from similar existing applications can also be a helpful source of candidate aggregation relations, as with other aspects of object models. As an aid to identifying all of the relevant parts, aggregates, and their relations, the analyst can ask whether identified classes have any components or participate in any aggregates that are relevant to the application. It is crucial to test any newly identified parts to determine whether there is any need to refer to them or their attributes separately. While much in-depth knowledge on the decomposition of certain objects may be available, there is no need to include it in the object model unless the components' attributes or operations must be distinguished from the whole object for some purposes of the application. The parts break-down of an airplane, for example, may be unnecessary for an application that is scheduling its missions, though essential for one that is scheduling its maintenance.

A.2.7 Identifying Other Associations

Definitions

The part-whole relations and inheritance relations just discussed are but two examples of the multitude of different types of associations that may hold between classes or objects. Examples of other types of associations include the common business associations of *Works-for, Manages*, *Responsible-for, Works-on,* and *Manufactures* from the simple business model illustrated in Figure A-6 on page A-15. Other examples, taken from a Battle Management/C3 information architecture developed by the Ballistic Missile Defense Organization [BMDO94], include the associations of *is_selected_by, monitors, defines, sets,* and *issues* between *Commander* objects and *Plans*, *Situations*, *Defended_Assets*, *Authorizations,* and *Orders*, respectively, as illustrated in the object model diagram of Figure A-11.

Associations are nothing new, having been widely adopted as a means of organizing data in databases, where they are better known as *relations*. However, while relations in relational databases may relate entities (objects) to the values of their attributes as well as to other entities, associations in OO systems are confined to relating objects or classes since attributes are separately identified. In OO systems, associations provide links between the associated objects, supporting their traversal both for retrieval of related information and access to the services and operations available from the associated objects. Whereas databases commonly use many-place relations,[3] OO systems tend to emphasize the use of binary associations (between two objects) because of their simplicity and because attribute

[3] A many-place relation is one that relates many attribute values and/or entities in each of its instances.

values are captured separately.

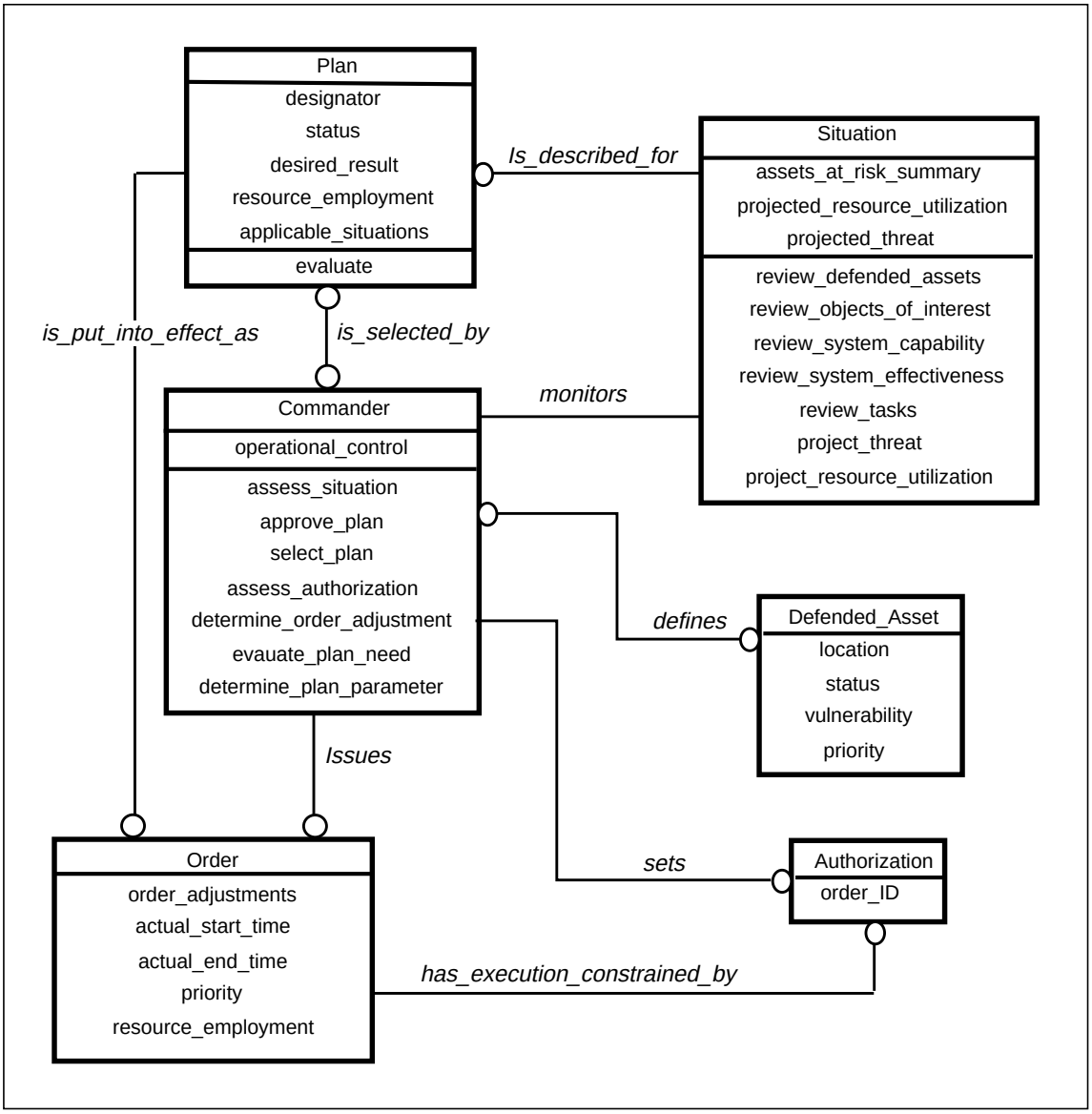


Figure A-11. Plan Generation and Selection Object Model Example

Diagram Notation

Associations admit a variety of different types of qualifiers, including cardinality, roles, attributes, and reification. The cardinality of an association constrains the numbers of instances of one class that can participate in this association with members of the other class. A few examples of diagramming conventions for indicating the different cardinalities of associations are presented in Figure A-5 on page A-14. Specific examples of binary associations between pairs of classes are illustrated in Figure A-6 on page A-15 and Figure A-

11 on page A-32, using the OMT object model notation. Ternary (3-place) and higher arity associations are represented with somewhat more complex links connecting multiple classes/objects. Roles may be specified for the different entities involved in an association, as *employee* is attached to *Person* and *employer* to *Company* in Figure A-6. Attributes may be attached to an association link to identify properties of the association, as *job title* is attached to the *Works-for* association in Figure A-6. An association may even be reified, or turned into an object, including operations as well as attributes, although this option is not discussed in many methodologies.

Identification Techniques

Associations may be identified from the usual sources of domain information: requirements statements, domain expert interviews, and scenario analysis. Sentences describing an application domain that use transitive verbs to relate a subject and object will often specify an association between them. The particular verb and context will, ordinarily, identify the nature of the association, whether it is an inheritance, part-whole, or some other type of association. For example, a sentence from a requirements statement or interviews for a C3 system may state something like the following:

> A commander selects plans, monitors situations, defines defended assets, and issues orders.

This sentence contains four transitive verbs and their objects, identifying four of the associations represented previously in Figure A-11.

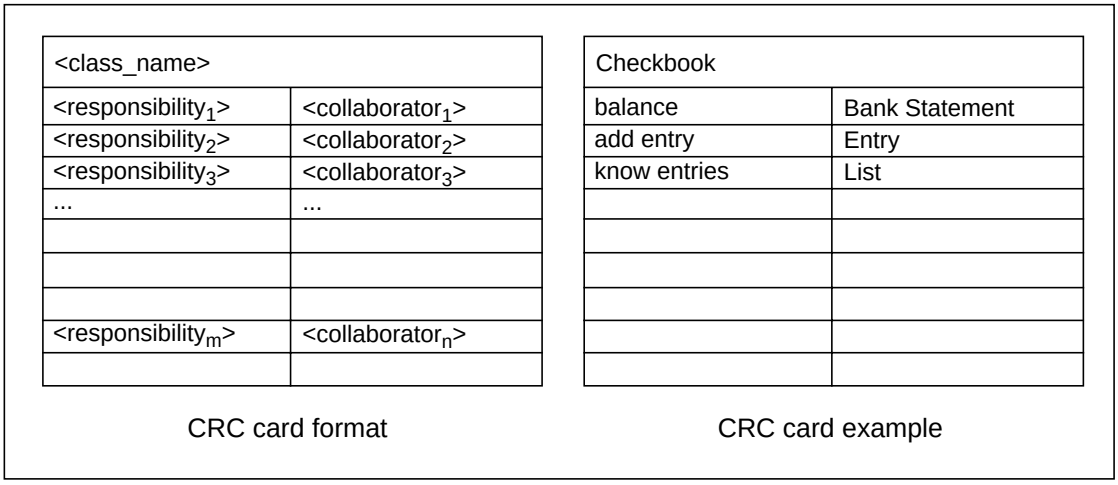
A.2.8 Maintaining a Data Dictionary

A data dictionary is an alphabetical listing of the types of data items in a system, with a definition or description of each one. In relational database systems, the data dictionary includes descriptions of the types of data items found in each column of each data table. In OO systems, a minimal data dictionary contains entries for classes and their attributes, which correspond roughly to tables and their columns in relational systems. In addition, some OO methodologies recommend storing other object information in the data dictionary. Rumbaugh's OMT recommends preparing a data dictionary during analysis with one paragraph for each class describing its restrictions, associations, attributes, and operations [RUMB91]. Booch recommends starting a data dictionary with the initial identification of classes and objects, and maintaining it as the central repository for all of the elements of the system, including attributes, responsibilities, associations, categories of classes, modules, and subsystems [BOO94a, pp. 236, 238, 243].

While the object/class diagrams developed during analysis and other diagrams developed in later stages of development may be simplified to focus upon the most important features of an OO system in development, a data dictionary should be an exhaustive compilation of every element in the system (among the types it covers). Ordinarily, good CASE tools will support a direct mapping between an electronic version of a data dictionary and the various OO modeling diagrams of whichever methodology is supported. A well-maintained and readily accessible data dictionary can be an effective tool in managing software development, providing detailed documentation of the models developed during analysis and subsequent stages of software development.

A.2.9 Performing CRC Card Analysis

Thus far, we have been discussing the identification of the individual elements of an object model: classes, attributes, operations, and associations. CRC card analysis, in contrast, is a technique for simultaneous development and refinement of multiple aspects of such models. CRC cards are pieces of paper or index cards, each of which is labeled starting at the top with a class name, followed by its responsibilities and its collaborators, as illustrated in Figure A-12. The name of the class on a card may be accompanied by a short description of its purpose if it is not obvious.[4] It is recommended that all writing on the cards be done in pencil since the analysis process is iterative and expected to require changes and adjustments to classes and their features as it proceeds [BOO94a, p. 159].



Source: [WILK94]

Figure A-12. CRC Card Layout

[4] Including class descriptions on CRC cards is recommended by Wirfs-Brock in her Responsibility-Driven Design methodology [HUTT94, p. 192].

The collaborators of a class are all the other classes that interact with it either by calling its services or having their services called by it. Tracing the interactions of collaborations actually falls into the domain of the dynamic modeling part of analysis. While CRC card analysis does include some dynamic modeling in virtue of its inclusion of collaborations, we introduce it here in object modeling because it can also function as an effective means of identifying classes and responsibilities. The widely noted blurring of distinctions between different stages of OO software engineering extends to the separate steps or activities of individual stages such as analysis. *The overlap between object modeling and dynamic modeling found in CRC card analysis is but one example of how many of the different aspects of OO analysis may proceed in parallel at any given time.*

CRC card analysis is essentially a scenario-driven activity in which the cards are used to represent the classes/objects involved in specific scenarios. The technique is designed to support collaborative analysis by groups of developers, possibly including users and domain experts. Cards may be laid out on a table or tacked up to a board, and grouped to represent patterns of collaboration, facilitating comprehension of and adjustments to the model by a whole group. The technique begins with a development team walking through each scenario, associating the required scenario activities with the responsibilities of specific classes and identifying the communication between collaborating classes as they call upon each other's services. As subsequent scenarios are analyzed, the CRC cards are adjusted to accommodate them. New classes may be added to handle new responsibilities, existing responsibilities may be reassigned to specializations or generalizations of existing classes, and new collaborations may be recorded. The process iterates until it converges upon a stable set of CRC cards that cover all the activities of all of the scenarios under consideration. The results are transferred to the object model and the dynamic model.

CRC card analysis was introduced by Beck and Cunningham as a pedagogical technique for teaching OO programming [BKCN89]. It has been adopted as one of the central techniques in Wirfs-Brock's Responsibility-Driven Design and in Hoeydalsvik's Object-Oriented Role Analysis and Modeling [HUTT94]. Booch has recommended it as "a simple yet marvelously effective way to analyze scenarios" [BOO94a]. While the technique has been criticized by Jacobson as potentially hard to scale up to large problems [JACO93, p. 499], it is not clear that this criticism is justified since the technique's application may be limited to manageable subsets of a large system's scenarios at any given time.

A.2.10 Identifying Constraints and Rules

Constraints are conditions that must be preserved by a system, such as cardinality constraints on associations and range constraints on the values of attributes. Rules may be considered as a type of (potentially complex) constraint of the form:

If <*condition_1, condition_2,... condition_n*> then <*result*>

in which their results are constrained to hold whenever their conditions hold. Alternatively, constraints may be considered a simple type of rule. The Object Management Group (OMG), for example, identifies *constraint rule* and *assertion rule* as types of rules in the basic concepts of its technical framework [HUTT94, pp. 124-125]. However their relationship is conceptualized, constraints and rules are distinctive features of many OO systems. Both express declarative conditions whose truth should be preserved by the system, in contrast with procedures, operations, or services which do not have truth values.

While practically all OO methodologies model cardinality constraints on associations, as described in the previous paragraph, other types of constraints are often neglected in object modeling. The OO analysis of Coad-Yourdon, for example, does not discuss modeling of either attribute constraints or rules [CDYD91]. If-then rules are not even mentioned explicitly by most OO methodologies, though they are essential components of OO systems developed in the field of artificial intelligence (AI). The typical “rule-based” or expert system in AI is an OO system with “frames” or “units” containing class object information along with related rules that trigger under specified conditions such as attribute updates. Despite such wide usage in AI, only 1 of the 21 different OO methodologies recently surveyed by the OMG explicitly included if-then rules in its object models [HUTT94]. Thus, most existing methodologies would have to model rules in some other way, such as objects or operations. As an object, a rule might be associated with domain objects whose attribute updates might trigger a rule method which fires or executes the rule. Alternatively, each rule might be encapsulated as an operation in a class whose attributes or associations are involved in its conditions or results.

Constraints are represented differently in object model diagrams, depending upon their type and the OO methodology being used. Cardinality constraints in associations are commonly represented by qualifiers, numeric or symbolic, on the ends of links, as discussed previously. Constraints on a class or its members are included within braces (curly brackets—{ }) on the class icon in Booch’s class model diagrams [BOO94a]. Rumbaugh’s OMT also places them within braces, although just outside his class icons [RUMB91].

Constraints on associations are also enclosed in braces and placed on or near the relevant association links in both Booch and OMT diagrams. If-then rules are included in the class icon on a par with attributes and operations in one methodology, the Graham/SOMA (Semantic Object Modeling Approach) [HUTT94].

A.2.11 Partitioning the Analysis Model

All but the simplest object models benefit from some organization or partitioning of their object classes into related groups. This aids comprehension during analysis and design, and provides a basis for formulating and implementing subsystems during design and implementation. Most OO methodologies prescribe some such partitioning of the object model, although different methodologies often prescribe different types of partitions at different stages of development using different terminology.

Jacobson, for example, insists that the object model be divided at the outset of object modeling into three basic groups of objects/classes: interface objects, entity objects, and control objects [JACO93, p. 132, 174-195]. Interface objects include the obvious things such as computer display windows, menus, and buttons, as well as other input/output devices such as display panels, instruments, and machinery. Entity objects include the natural entities in the application domain, such as the employees, companies, and departments of a business application, as well as other persistent objects. Control objects are sometimes needed for behaviors that do not fall naturally into the other types of objects.

This *a priori* division of the object model into class/object types is accompanied by a further division into *subsystems* based upon functional groupings of identified classes/objects. Subsystem groupings are included as part of the analysis model by Jacobson. According to Jacobsen, the groupings may not be delineated until the end of analysis in small projects, although larger projects may need to identify subsystems much earlier in order to distribute development work in accord with this partition of the system [JACO93, pp. 195-199]. For the design phase, Jacobson develops another partitioning concept, the *block,* which is an abstraction consisting of one or more analysis-level classes/objects.

In the Booch method, partitioning of classes/objects is not pursued during the analysis phase. Classes are grouped into *categories* during architectural design, and the program-level modules that implement classes are grouped into *subsystems* during design [BOO94a, p. 244]. Booch distinguishes logical-design groupings as *categories* from physical design groupings as *subsystems*; Jacobson does not, referring to analysis, design, and implementation groupings as *subsystems*.

Rumbaugh's OMT suggests grouping classes and associations into relatively small *modules* during the analysis phase. OMT recommends keeping each module small enough so that its object model diagram can fit on a single *sheet* of paper [RUMB91, p. 43]. OMT postpones establishing major partitions of the object model until the design phase, where a hierarchy of *subsystems* is developed that includes modules as the lowest level of subsystem [RUMB91, pp. 219-220]. Naturally, the composition and organization of the modules from the analysis phase may have to be adjusted during design to better accommodate the constraints of the implementation environment.

Coad-Yourdon's analysis phase partitions classes and their features into coherent *subjects* which are comparable to what are called *subsystems* by Jacobson and Rumbaugh, and *categories* by Booch [CDYD91, pp. 106-118]. In addition to a partitioning into *subjects*, there is an orthogonal partitioning of an OO system into *layers* and *components* in Coad-Yourdon's OOA, as illustrated in Figure A-13 [CDYD91, p. 179]. Among these OO analysis layers, only the *subject* layer represents any real partition of the object model, assigning different classes/objects to different subject groupings. The other layers simply view the object model through different sets of its features (e.g., attributes or services), while leaving the classes/objects unpartitioned. This OO analysis division into *components* is comparable to Jacobson's division into interface, entity, and control objects, although Coad-Yourdon's OOA further divides control objects into task control and data control (or management) objects.

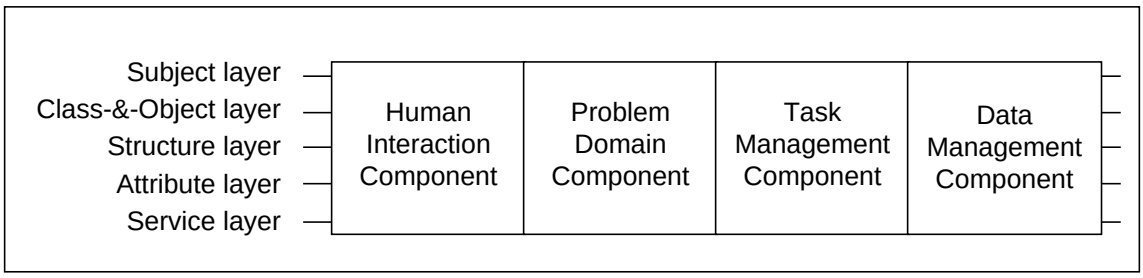


Figure A-13. Coad-Yourdon's Multi-Layer, Multi-Component Model

The Shlaer-Mellor method also insists on a high-level division of the system and its models into distinct subject areas that it calls *domains*, comparable in scope to the components of Coad-Yourdon's OOA. Shlaer-Mellor identifies four types of domains: a single *application domain*, *general domains* (including interfaces to users and instruments), an *architectural domain* (including data management, multi-tasking, and distributed processing), and *implementation domains* (including operating systems and

programming languages) [HUTT94, p. 167]. Partitioning at this level is listed as the first step of this development methodology. The method also allows that domains may be "further partitioned into *subsystems* to provide coherent and manageable pieces of work" [HUTT94, p. 166].

In summary, we observe that some methodologies prescribe an *a priori* partitioning of their models beginning in the analysis phase, as illustrated by the partitions listed in Table A-1. In addition, most methodologies prescribe an *a posteriori* partitioning of their object models into closely related groups of objects/classes, though they often use different terms to describe such groupings and to introduce them at different stages of development. Jacobson, Rumbaugh, and Shlaer-Mellor all refer to these partitions as *subsystems;* Coad-Yourdon call them *subjects*; and Booch refers to logical-design partitions as *categories*, while his physical-design partitions are called *subsystems*.

Table A-1. Different Types of A Priori Partitions of OO Models

Methodology	Type of Partition	Partitions		
Jacobson's (Objectory)	Object Types	Interface	Entity	Control
Shlaer-Mellor Method	Domain	General	Application	Architectural
Coad-Yourdon OOA	Components	Human Interaction	Problem Domain	Task Management
Rumbaugh's OMT	[none]			
Booch Method	[none]			

Some grouping of the classes in an object model is practically essential in any substantial OO project and is widely recommended. No specific a priori *partition is necessary, though some such partition can be helpful—hence most developers may as well conform with their chosen methodology.*

A.3 DYNAMIC MODELING

The dynamics, or change over time, of a system and its components are the subject of dynamic modeling. Within OO systems, interest in dynamics is naturally focused on the dynamics of objects: the possible changes to individual objects over time and the interactions between objects over time. Thus, models of such dynamic behavior ordinarily divide into two types: state transition models of individual objects and state interaction models for groups of objects. OO methodologies commonly use both types of dynamic models to capture the intended dynamic behavior of a system, although they are divided about where to start such modeling. Some place it within analysis, while others delay it until design.

The process of dynamic modeling may be decomposed into a set of activities culminating in the completion of the two types of dynamic models. One such decomposition, provided by OMT [RUMB91], is as follows:

- Prepare scenarios.
- Identify events.
- Develop event trace diagrams [interaction diagrams] for each scenario.
- Prepare an event flow diagram for the whole system.
- Build state diagrams for each dynamic class.

The Booch method proposes a similar sequence of activities for dynamic modeling during the analysis phase [BOO94a, pp. 253-254]:

- Identify primary function points of system and group.
- Storyboard scenarios for groups of function points (CRC cards recommended).
- Document scenarios using “object diagrams” (or interaction diagrams).
- Generate secondary scenarios, as needed, for exceptional conditions.
- Develop finite state machines (state diagrams) for dynamic classes of objects.

OMT, the Booch method, and the Shlaer-Mellor method all place such dynamic modeling activities within the analysis phase of development. Jacobson, in contrast, initiates use case (or scenario) modeling in his analysis phase but postpones building dynamic models until design.

Regardless of what phases these activities are relegated to, the essence of typical approaches to OO dynamic modeling lies in the use of scenario analysis to develop two types of dynamic models: interaction diagrams (also called event trace diagrams or object diagrams) and state transition diagrams.

A.3.1 Preparing Scenarios

Dynamic analysis is ordinarily initiated with some sort of scenario preparation in which different scenarios of alternative uses of the system are prepared to document the system's required dynamic behaviors. A scenario is a (partially) ordered sequence of events that illustrates a single type of use of the system. The concept of a scenario found in [BOO94a] and [RUMB91] is essentially identical to that of the "use case" used by Jacobsen [JACO93], although the latter concept is embellished by its associated use case diagrams and an emphasis on users, as discussed previously.

Different techniques have been proposed for preparing scenarios. Rumbaugh suggests examining the problem statement for required interaction sequences. Booch suggests clustering sets of related function points in the system to identify scenarios that exercise the behaviors specified by the function points.[5] CRC card techniques, as described previously in Section A.2.9, are often recommended for storyboarding scenarios for dynamic analysis, as well as for identifying objects/classes in building an object model.

Since each scenario represents a single use of the system, multiple scenarios are ordinarily necessary to capture alternative courses of action in user-system interactions. One common approach to covering such alternatives divides scenarios into primary ones which cover typical operating conditions, and secondary ones which cover exceptional cases. Using this approach, both Booch and Jacobson recommend identifying the primary scenarios first, and supplementing them with the exceptions as needed.

Several authors emphasize the importance of including domain experts in scenario analysis in order to ensure accuracy and coverage. Booch also recommends including quality assurance personnel since scenarios represent testable behavior around which some testing may be organized.

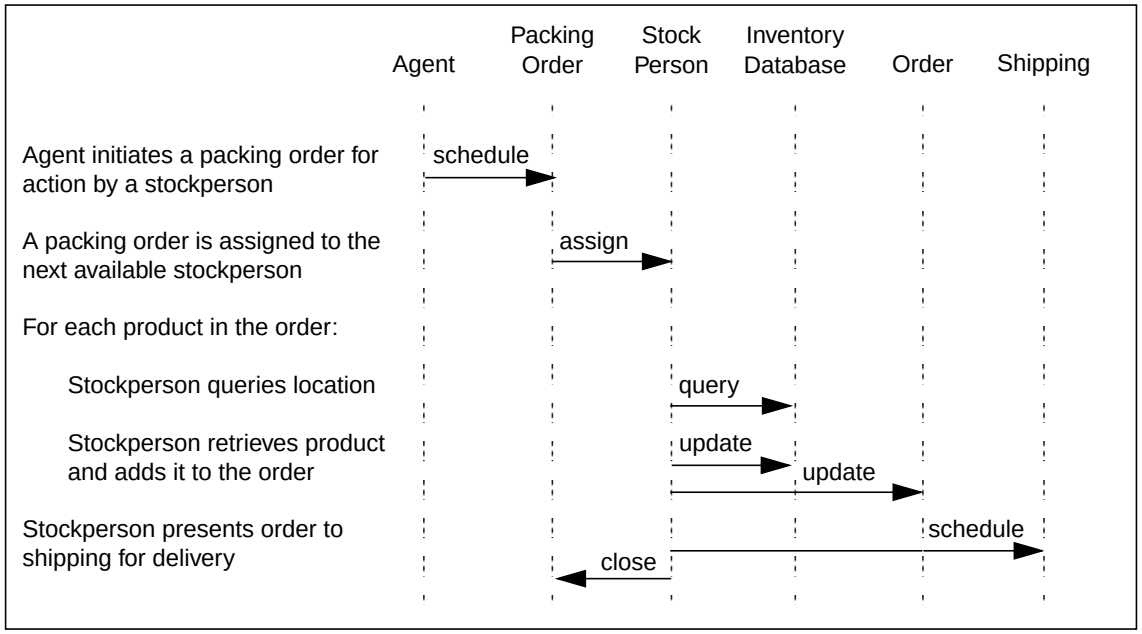
---

[5] Function points represent the observable and testable behaviors of a system, commonly in response to some external event [BOO94a, p. 252-254].

A.3.2 Generating Interaction Diagrams

When a scenario is first developed, it is often expressed by a list of natural language sentences describing the sequence of activities involved. Transforming such scenario descriptions into the activities of an OO system involves identifying the specific activities or events and mapping them to the interactions of objects in the system. This transformation is facilitated by a variety of techniques, such as textual analysis of scenario descriptions to identify activities, and use of CRC card techniques to directly represent a scenario via the collaborations between the involved objects. The results of this transformation are readily expressed in an interaction diagram which displays the interaction between all of the objects of a specific scenario.

Figure A-14 shows an interaction diagram for a simple order packing scenario in

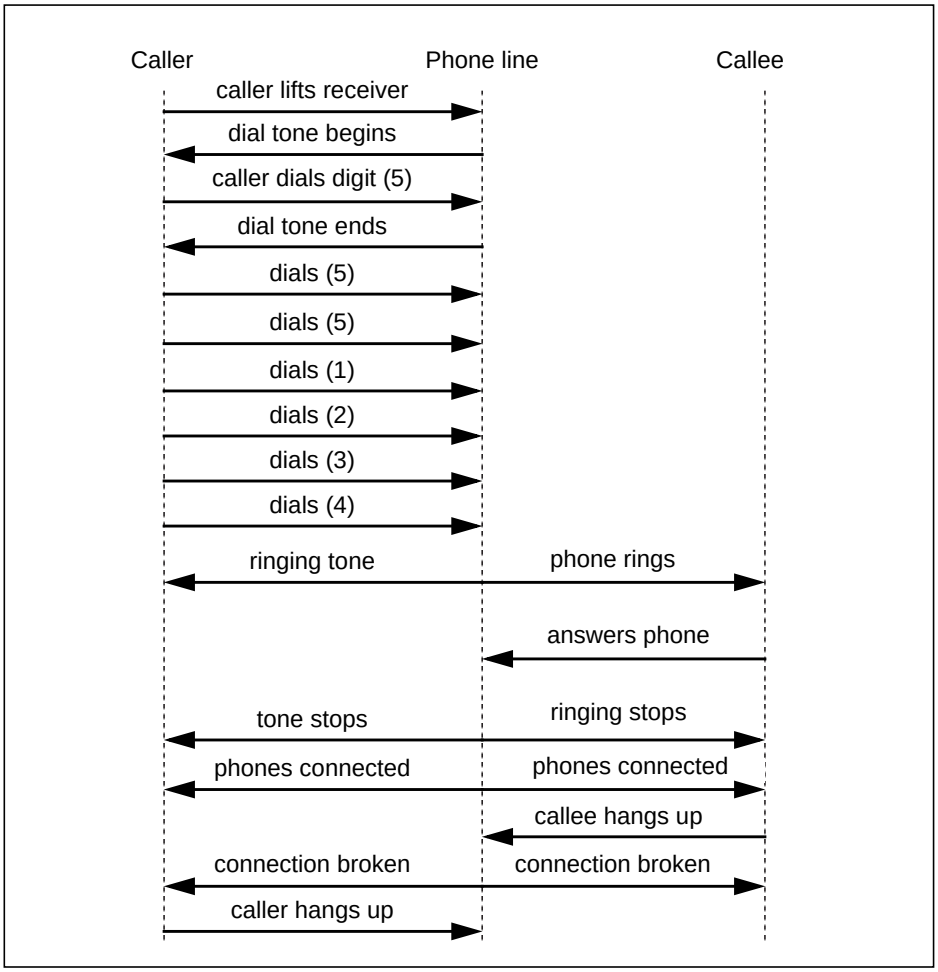


Source: [BOO94a, p. 388]

Figure A-14. Interaction Diagram for an Order Packing Scenario

which an order of goods is packed by a stockperson. While the diagram uses Booch's drawing conventions, these conventions are very close to those used in Jacobson's interaction diagrams and OMT's event trace diagrams. An example using OMT's notation of interactions among the objects involved in a phone call is shown in Figure A-15. In all cases, vertical lines representing specific objects are labeled with the type of the object at the top; horizontal arrows represent interactions between objects, directed from the client object to

the server object; labels on the arrows indicate the type of service requested by the client object; and interactions are listed in temporal order from the top to the bottom of the diagram. Booch and Jacobson optionally describe the individual interactions in a column to the left, as shown in Figure A-14.



Source: [RUMB91, p. 87]

Figure A-15. Event Trace Diagram for a Phone Call

Booch discusses two types of diagrams for modeling dynamic behavior between objects in a scenario. His "interaction diagram" in Figure A-14 is described as a generalization of OMT's event trace diagrams and Jacobson's interaction diagrams [BOO94a, p. 217]. His other type of dynamic diagram, which he calls an "object diagram," is his original approach for representing such dynamics using a graph of cloud-like Booch object icons connected by arcs displaying their interaction events. These object diagrams are entirely distinct from the object/class models discussed previously, which are purely static models. Both Booch's interaction diagrams and his object diagrams have very similar information

content but in different forms. We use interaction diagrams for illustrative purposes because their design better reveals the temporal flow of the events in a scenario, and because they are employed by more methodologies.

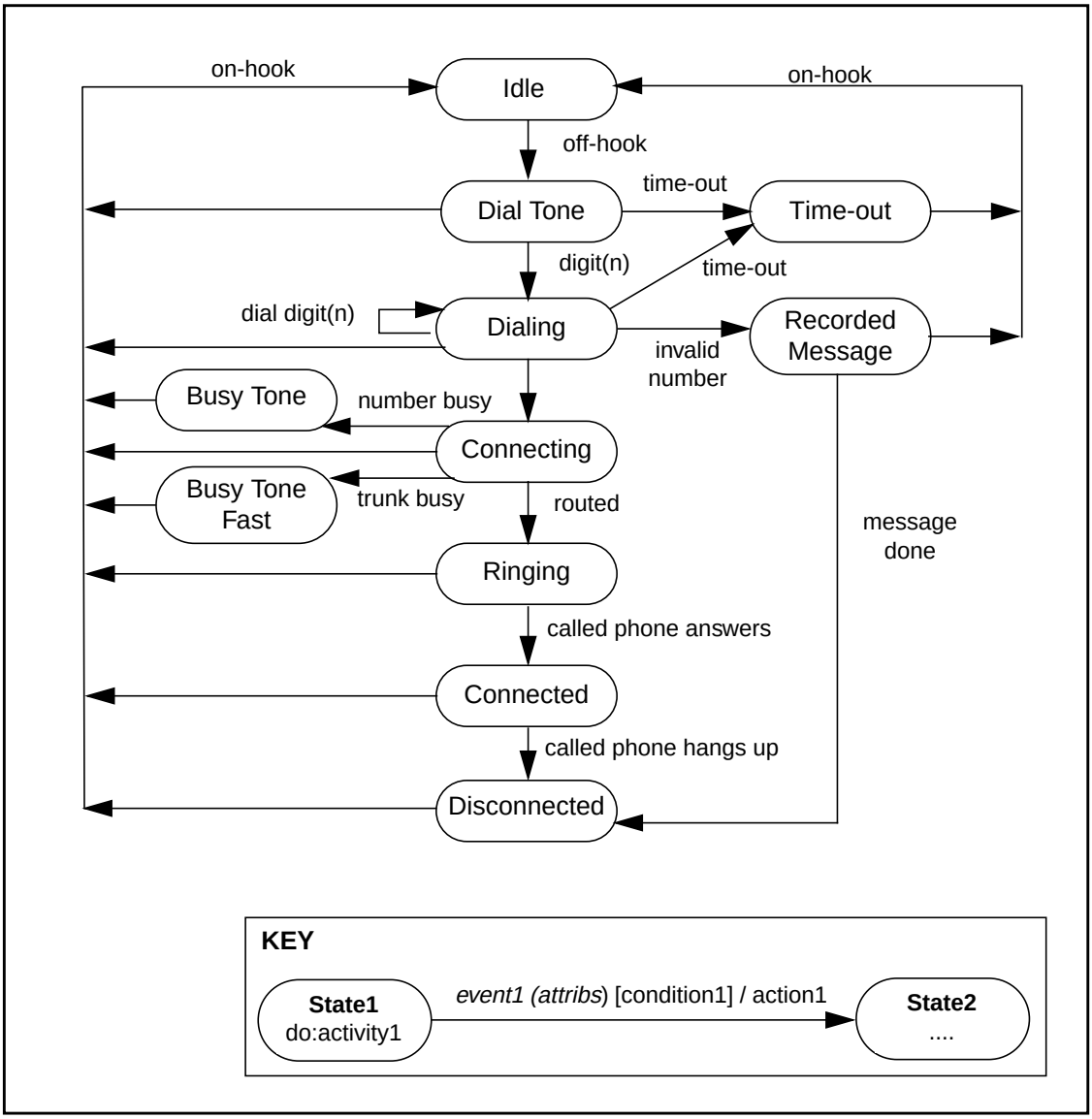
Another type of representation for the dynamics of scenario interactions is the "event flow diagram" used by OMT to display interactions at a higher level between groups of objects (such as modules). In these diagrams, modules are represented by rectangles that are connected with directed arcs between them labeled with their interactions. The sequence of operations, however, is not identified. While such high-level event flow diagrams may aid in organizing modules, the interaction diagrams provide an essential basis for determining the state transition requirements of individual objects. Every event with which an object is involved (as either client or server) on an interaction diagram may correspond to a transition on its state transition diagram. Different scenarios typically generate overlapping sets of such transitions; these sets must be combined into one coherent set of possible transitions for each object class whose transitions are sufficiently complex.

A.3.3 Generating State Transition Diagrams

When the interaction dynamics of the objects in a class are sufficiently complex, OO analysis and design benefit from modeling the objects as finite state machines whose state changes over time in response to its interactions. Finite state machine models are commonly represented by state transition graphs in which the states of an object (or subsystem) are represented by icons, and the transitions between their states are represented by directed arcs labeled with the transition events, as illustrated in Figure A-16.

The illustrated state transition diagram is taken from Rumbaugh [RUMB91] who, along with Booch and others, adapts the notation developed by Harel [HARL87]. It shows the different states of a phone line—Idle, Dial Tone, Dialing, Connecting, Ringing, etc.—and the events that are required to transition between them. Other elaborations on arcs and states are described by the key, though they are not needed for this example. While there are substantial differences in other methodologies in the drawing conventions used for such state diagrams, the basic content is practically the same.

The states in state transition diagrams for OO modeling are the states of the object being modeled, which are understood as an abstraction of some range of values of the model's attributes and associations. A given state might correspond to a particular range of one or more attribute values, as a bank account state of being *overdrawn* can correspond to a negative amount in a *balance* attribute. Alternatively, special attributes may be created for



Source: [RUM91 p. 90]

Figure A-16. A State Transition Diagram for a Phone Line

a class to identify some or all of its states. Different states need be distinguished in OO modeling only insofar as they determine different responses to events. Events in an OO modeling context are occurrences of requests for services from one object to another. Thus, it is only when an object's services or operations respond differently, depending on its condition, that different states must be distinguished.

State transition diagrams represent the complete dynamic behavior of a single object for all the scenarios in which it may participate. Interaction diagrams, in contrast, show the dynamic behavior of a group of objects for just a single scenario. Both types of

dynamic modeling diagrams are helpful in determining how an object's operations should respond to different sequences of events. Each event in an interaction or state diagram represents a request for a service (or a call of an operation), and the response may often depend on the prior sequence of events. Thus, modeling such sequences using multiple scenarios and interaction diagrams helps ensure that the full breadth of intended behavior is captured for those involved objects.

If object interactions are simple enough and the responses of specific classes do not vary much within a scenario or between different scenarios, it may be straightforward to proceed directly to algorithm design for the operations involved. However, when there is substantial variation and context dependency in the operational responses, an intermediary step of detailed modeling of individual object dynamics can be very helpful. State transition diagrams provide such detailed models, integrating all the dynamics of the different scenarios in which an object participates. Figure A-16 illustrates how the state transition diagram for a phone line captures all the possible events in which it participates for the event trace diagram of Figure A-15, as well as for others not illustrated there. The transitions to different types of busy signals shown on the phone line state diagram, for example, appear on the state diagram but not in the normal phone connection event trace diagram of Figure A-15.

State transition diagrams may be used at different levels of granularity and during different phases of development to model subsystems, blocks, modules, or individual objects. Booch recommends developing state transition diagrams during analysis for the system as a whole, and delaying dynamic analysis of individual classes or collaborations of classes until the design phase [BOO94a, p. 200]. Jacobson postpones his use of state transition graphs until design, and focuses their application at the level of blocks which consist of one to several closely related classes [JACO93, pp. 233-241]. Rumbaugh focuses on modeling individual classes with state diagrams during dynamic modeling in his analysis phase. However, he allows that these models, along with the other analysis models, may be optimized, refined, or extended during the design phase [RUMB91, p. 264].

State transition diagrams are widely recommended for analyzing just those classes, blocks, and subsystems whose dynamic behavior is complex enough to benefit from the clarification they provide for design and implementation of their context-dependent operations.

It is possible to do effective OO development without any use of state transition diagrams, as evidenced by the existence of OO methodologies, such as Wirfs-Brock's Responsibility-Driven Design [HUTT94], which do not include them anywhere within their lifecycle. However, as the interactions in a system become more complex, it also becomes increasingly difficult to design and implement context-dependent operations without the aid of a detailed general dynamic model which may be constructed using state transition diagrams.

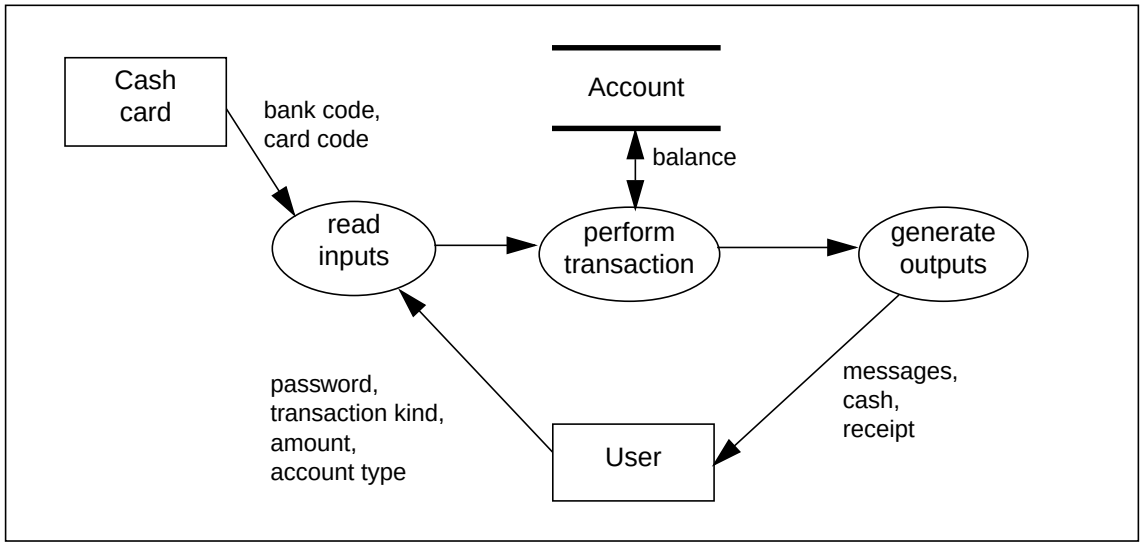
A.4 FUNCTIONAL MODELING

The essence of functional modeling is the modeling of system functions and their effects. System functions can be modeled at multiple levels from the overall primary functions of a system to the detailed functional transformations of individual pieces of data. Indeed, it is the successive decomposition of system functions into their constituent functions that forms the heart of traditional structured analysis and design techniques. While there is nothing object oriented about functional modeling, some OO methodologies incorporate some functional modeling among their modeling techniques.

Informal functional modeling exists in many OO methodologies under the guise of activities such as “key mechanism” analysis [BOO94a], scenario analysis [RUMB91], or use case analysis [JACO93]. Such analyses tend to be fairly high-level characterizations of system functionality, commonly expressed in natural language. These high-level functional analyses provide an effective means of initial expression of system requirements which are ordinarily quickly transitioned to OO models in terms of objects, their attributes, and their services. At the other end of the spectrum of level-of-detail are detailed functional models of the algorithms used in implementing specific object operations. This functional modeling is also common in OO development methodologies where it typically appears either late in the design stage or during implementation. However, neither of these types of functional modeling is the focus of what is ordinarily meant by “functional modeling” when it appears as a separate activity within an OO development methodology. Rather, the functional model is commonly understood to refer to “multiple data flow diagrams which show the flow of values from external inputs, through operations and internal data stores, to external outputs” [RUMB91, p. 123].

Data flow diagrams model the flow of data through a system as it is transformed by processes that operate upon it. An example adapted from Rumbaugh [RUMB91] is presented in Figure A-17. This diagram uses a common data flow notation wherein external entities are represented by labeled boxes, data flows are illustrated by labeled arrows, processes are shown as elliptical icons, and data stores appear between horizontal bars. Functional modeling is pursued by decomposing higher-level functions and generating the corresponding lower-level data flow diagram for each non-primitive process. The data flow diagrams may then be augmented by descriptions of each function or process in the diagrams, and of any constraints on data values or processes.

Data flow models like the one displayed in Figure A-17 can be used within an OO development to aid the identification of objects, their attributes, and their services. Meth-



Source: [RUMB91, p. 181]

Figure A-17. Data Flow Diagram for an Automated Teller Machine

odologies, like OMT, that incorporate data flow diagrams offer guidance on extracting object information from them. It is obvious that processes should map into objects services and data stores should map into the object attributes. But determining which objects map to which processes and data stores is not a trivial procedure. Some of the procedures for handling this transition are discussed in a companion report on reengineering systems strategies [IDA95d].

Among OO methodologies there is a wide variation on the extent of functional modeling that they prescribe. Some, like Booch, intentionally limit the extent of functional modeling in order to avoid "polluting the design with preconceived algorithmic notions" [BOO94a, p. 161]. Others, such as OMT and the Shlaer-Mellor method, prescribe functional analysis of the system using a series of data flow diagrams. Whether data flow diagrams are a help or a hindrance in OO development remains debatable.

Recently, Rumbaugh has clarified his position on his usage of data flow diagrams, as follows:

> There has been a good deal of misunderstanding regarding the role of data flow diagrams in the OMT methodology. We never intended that one should perform an SA/SD-like data flow analysis of a system. The functional model is intended to show the dependencies between values in the system. It is usually best expressed in equations, as I have done above. On the other hand, it is often useful to be able to see the relationships between the various

> functions and values, and a data flow diagram can show the relationships graphically, just as an object diagram shows structural relationships. [RUMB93, p. 18].

More recently still, Rumbaugh has acknowledged that "trying to integrate data flow diagrams with purely object-oriented models just doesn't work very well" [BOO94b, p. 3].

Some OO methodologies, such as Information Engineering \with Objects (IE\O) [HUTT94, pp. 85-92], include functional data flow models as optional components of their analysis model. Even Booch allows the data flow diagrams of structured analysis may be used as a front end to OO design, though he warns that "our experience indicates that using structured analysis as a front end to object-oriented design often fails when the developer is unable to resist the urge of falling back into the abyss of the structured design mindset" [BOO94a, p. 161]. Simply adding functional modeling to object and dynamic modeling during analysis should carry less risk than its exclusive use. But it is not clear that such an addition would contribute much of any value to the analysis products. If scenario analysis has already been performed carefully, then any functionality that would be covered by a data flow diagram should already be covered in the dynamic model, where it is tightly integrated with the objects of the object model. Separate functional modeling could act as a check on the other models, but it is not clear that this is the best approach to such checking, especially if it threatens to distort the OO design.

One situation in which the functional models characteristic of structured analysis can be helpful arises when they already exist as part of the documentation of a legacy system that is being modernized. In such situations, legacy data flow diagrams may aid the reverse engineering of system requirements and the identification of scenarios and objects from the outset. When such a legacy system is being ported to an OO system, the migration system development still requires object and dynamic modeling, whether these are considered part of analysis or design. Implementing an OO system without first developing models of object structure and behavior would be merely "hacking" code, not "engineering" software. While functional models of legacy systems can aid understanding of requirements, they can never substitute for the OO models essential to OO software engineering. Further discussion of transitioning from the functional models of structured analysis to OO analysis and design is presented in a companion report [IDA95d] where a variety of reengineering strategies are described.

Given the widely acknowledged problems in incorporating data flow diagrams into OO development and the redundancy of any information they might provide, we recommend against their usage unless they already exist, whence they may be used to aid OO modeling, though at some peril of "falling back into the abyss of the structured mindset."

LIST OF REFERENCES

[ANSI92] American National Standards Institute, ANSI X3.135-1992, *Database Language SQL*, New York, NY, 1992.

[ARNL94] T. R. Arnold and W. A. Fuson, "Testing 'In A Perfect World'," *Communications of the ACM,* Vol. 37, No. 9, September 1994, pp. 78-86.

[BARB94] S. Barbey, M. Ammann, and A. Strohmeier, *Open Issues in Testing Object-Oriented Software, Technical Report No. 94/45*. Departement D'Informatique, Swiss Federal Institute of Technology—Lausanne, Switzerland, 1994.

[BIND94a] R. V. Binder, "Object-Oriented Software Testing: Introduction," *Communications of the ACM,* Vol. 37, No. 9, September 1994, pp. 28-29.

[BIND94b] R. V. Binder, "Design for Testability in Object-Oriented Systems," *Communications of the ACM,* Vol. 37, No. 9, September 1994, pp. 87-101.

[BKCN89] K. Beck, and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *SIGPLAN Notices,* Vol. 24, No. 10, 1989.

[BMDO94] Ballistic Missile Defense Organization, *National Missile Defense Battle Management, Command, Control and Communications Domain Information Architecture*, Washington, DC, 25 October 1994.

[BOO94a] G. Booch, *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994.

[BOO94b] G. Booch, "The Booch Method: Scenarios," *Report on Object Analysis and Design*, Vol. 1, No. 3, September-October 1994.

[CATT93] R. G. G. Cattell, editor, *The Object Database Standard: ODMG-93*, Morgan Kaufmann, San Mateo, CA, 1993.

[CATT94] R. G. G. Cattell, *Object Data Management: Object-Oriented and Extended Relational Database Systems,* Addison-Wesley, Reading, MA, 1994.

[CDYD91] P. Coad and E. Yourdon, *Object-Oriented Analysis,* 2nd edition, Yourdon Press, Englewood Cliffs, NJ, 1991.

[DOD88] Department of Defense, DOD-STD-2167A, *Defense System Software Development*, Washington, DC, U. S. Government Printing Office, February 29, 1988.

[DOD90] Department of Defense, *A Plan for Corporate Information Management for the Department of Defense*, forwarded by the Executive Level Group for Defense Corporate Information Management, September 11, 1990.

[DOD91] Department of Defense, *Software Technology Strategy*, prepared for the Director of Defense Research and Engineering (DDR&E) in partial fulfillment of the DDR&E Software Action Plan, December 1991.

[DOD92a] Department of Defense, MIL-STD-SDD (Draft), *Software Development and Documentations*, December 22, 1992.

[DOD92b] Department of Defense, DoD 8020.1-M (Draft), *Functional Management Process for Implementing the Information Management Program of the Department of Defense*, August 1992.

[DOD93a] Department of Defense, DoD Directive 8120.1, *Life-Cycle Management (LCM) of Automated Information Systems (AISs)*, January 14, 1993.

[DOD93b] Department of Defense, DoD Instruction 8120.2, *Automated Information System Life-Cycle Management Process, Review, and Milestone Approval Procedures*, January 14, 1993.

[DOD93c] Department of Defense, Defense Information Systems Agency, Center for Architecture, *Technical Architecture Framework for Information Management*, Version 2.0, November 1, 1993.

[DOD93d] MIL-STD-498 (SDD) *Software Development and Documentation*, presentation by Raghu Singh of The DoD Software Harmonization Working Group, December 9, 1993.

[DOD93e] Department of Defense, Memorandum: From Director of Defense Information. Subject: Interim Management Guidance on the Technical Architecture for Information Management, January 15, 1993.

[DOD94a] Department of Defense, MIL-STD-498, *Software Development and Documentation*, December 5, 1994.

[DOD94b] Department of Defense, *Data Item Descriptions (DIDs) for MIL-STD-498*, December 5, 1994.

[DOD94c] Department of Defense, DOD-STD-7935A (Revision A), *Automated Information Systems (AIS) Documentation Standards*, December 5, 1994.

[HARL87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, 8, 1987.

[HARR93a] Harris Data Services Corporation, *Software Development Plan (SDP) for the Base Level System Modernization*. Contract No. F01620-88-D-0086, CDRL Sequence No. A056, prepared for Standard Systems Center (AFCC), Director of Contracting, Maxwell AFB - Gunter Annex, AL, Harris Data Services Corporation, Montgomery AL, October 1993.

[HEND94] B. Henderson-Sellers, R. Jordan Kreindler, S. Mickel, "Methodology Choices—Adapt or Adopt?," *Report on Object Analysis and Design*, Vol. 1, No. 4, November/December 1994, pp. 26-29.

[HRTY93] P. Harmon, D. Taylor, and W. Morrissey, *Objects in Action: Commercial Applications of Object-Oriented Technologies*, Addison-Wesley, Reading, MA, 1993.

[HUTT94] A. T. F. Hutt, editor, *Object Analysis and Design: Description of Methods*, John Wiley & Sons, New York, NY, 1994.

[IDA93a] K. Jordan et al., *An Assessment of the Potential Implementation of Object-Oriented Technology in the Department of Defense*, IDA Paper P-2904, Institute for Defense Analyses, Alexandria, VA, October 1993.

[IDA95a] B. A. Haugh, M. C. Frame, and K. Jordan, *An Object-Oriented Development Process for Department of Defense Information Systems*, IDA Paper P-3142, Institute for Defense Analyses, Alexandria, VA, July 1995.

[IDA95b] D. Smith, B. Haugh, K. Jordan, *Object-Oriented Programming Strategies for Ada*, IDA Paper P-3143, Institute for Defense Analyses, Alexandria, VA, July 1995.

[IDA95c] B. Haugh, A. Noor, D. Smith, K. Jordan, *Legacy System Wrapping for Department of Defense Information System Modernization*, IDA Paper P-3144, Institute for Defense Analyses, Alexandria, VA, July 1995.

[IDA95d] K. Jordan, B. Haugh, *Software Reengineering Using Object-Oriented Technology*, IDA Paper P-3145, Institute for Defense Analyses, Alexandria, VA, July 1995.

[INTR93] Intermetrics, Inc., *Introducing Ada 9X: Ada 9X Project Report*, prepared for the Office of the Under Secretary of Defense for Acquisition, Washington, DC, 1993.

[JACO93] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA, 1993.

[LIPP91] S. B. Lippman, *C++ Primer*, 2nd edition, Addison-Wesley, Reading, MA, 1991.

[LOOM95] M. E. S. Loomis, *Object Database: The Essentials*, Addison-Wesley, Reading, MA, 1995.

[MCGR94] J. D. McGregor and T. D. Korson, "Integrating Object-Oriented Testing and Development Processes," *Communications of the ACM*, Vol. 37, No. 9, September 1994, pp. 59-77.

[MRPH94] G. C. Murphy, P. Townsend, and P. S. Wong, "Experiences With Cluster and Class Testing," *Communications of the ACM*, Vol. 37, No. 9, September, 1994, pp. 39-47.

[NEWB95] G. A. Newberry, "Changes from DOD-STD-2167A to MIL-STD-498," *CrossTalk*, Vol. 8, No. 4, April 1995, pp. 4-7.

[PSTN94] R. M. Poston, "Automated Testing from Object Models," *Communications of the ACM*, Vol. 37, No. 9, September 1994, pp. 48-58.

[RUMB91] J. Rumbaugh, M. Blaha, W. Premerlani, E. Frederick, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[SIGS95] SIGS Conferences, "Booch & Rumbaugh on Tour: The Evolution of Object Methods," (advertisement) *Object Magazin*e, Vol. 4, No. 9, February 1995, p. 49.

[TAYL92] D. A. Taylor, *Object-Oriented Information Systems: Planning and Implementation*, John Wiley & Sons, New York, NY, 1992.

[TOPR95] A. Topper, "Methodology: Object Technology '95," *Object Magazine*, Vol. 4, No. 9, February 1995, pp. 20-22.

[WILK94] Nancy Wilkinson, "An Informal Introduction," *Report on Object Analysis and Design*, Vol. 1, No. 4, November/December 1994, pp. 41-43.

GLOSSARY

Words used in the definition of a glossary term and that are defined elsewhere in the glossary are in **bold**.

Abstraction — Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties [RUMB91].

AIS Program — A directed and funded AIS effort, to include all **migration** systems, that is designed to provide a new or improved capability in response to a validated need [DOD93a].

Architecture — The organizational structure of a system or **CSCI**, identifying its components, their interfaces, and a concept of execution among them [DOD94a].

Automated Information System (AIS) — A combination of **computer hardware** and computer **software**, data, and/or telecommunications that performs functions such as collecting, processing, transmitting, and displaying information. Excluded are computer resources, both hardware and software, that are either physically part of, dedicated to, or essential in real time to the mission performance of weapon systems; used for **weapon system** specialized training, simulation, diagnostic test and maintenance, or calibration; or used for research and development of weapon systems [DOD93a]. However, as used here, AISs include systems for C2I, C3I, and C4I, even though they may be essential in real time to mission performance.

Class — A class can be defined as a description of similar **objects**, like a template or cookie cutter [NEL91]. The class of an object is the definition or description of those attributes and behaviors of interest.

Collaboration A request from a client to a server in fulfillment of a client's responsibilities [HUTT94, p. 192].

Commercial-off-the-Shelf (COTS) Commercial items that require no unique government modifications or maintenance over the life cycle of the product to meet the needs of the procuring agency [DOD93a].

Computer Hardware Devices capable of accepting and storing computer data, executing a systematic sequence of operations and computer data, or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control, or other logical functions [DOD94a].

Computer Program A combination of computer instructions and data definitions that enable **computer hardware** to perform computational or control functions [DOD94a].

Computer Software Configuration Item (CSCI) An aggregation of software that satisfies an end use function and is designated for separate configuration management by the acquirer. CSCIs are selected based on tradeoffs among software function, size, host or target computers, developer, support concept, plans for reuse, criticality, and interface considerations need to be separately documented and controlled, and other factors [DOD94a].

Contract The list of requests that a client **class** can make of a server class. Both must fulfill the contract: the client by making only those requests the contract specifies, and the server by responding appropriately to those requests [HUTT94, p. 192].

CRC Cards **Class-Responsibility**-Collaborator Cards. CRC cards are pieces of paper divided into three areas: the **class** name and the purpose of the class, the responsibilities of the class, and the collaborators of the class. CRC cards are intended to be used to iteratively simulate different scenarios of using the system to get a better understanding of its nature [HUTT94, p. 192].

Database A collection of related data stored in one or more computerized files in a manner that can be accessed by users or computer programs via a database management system [DOD94a].

Database Management System — An integrated set of computer programs that provide the capabilities needed to establish, modify, make available, and maintain the integrity of a database [DOD94a].

Encapsulation — . . . (also **information hiding**) consists of separating the external aspects of an **object**, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects [RUMB91]. The act of grouping into a single object both data and the operation that affects that data [WIR90].

Framework — A collection of **class** libraries, generics, design, scenario models, documentation, etc., that serves as a platform to build applications.

Government-off-the-Shelf (GOTS) — Products for which the Government owns the data rights, that are authorized to be transferred to other DoD or Government customers, and that require no unique modifications or maintenance over the life cycle of the product [DOD93b].

Inheritance — Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship [RUMB91]. Subclasses of a **class** inherit the operations of the parent class and may add new operations and new instance variables. Inheritance allows us to reuse the behavior of a class in the definition of new classes [WEG90].

Information Hiding — Making the internal data and methods inaccessible by separating the external aspects of an **object** from the internal (hidden) implementation details of the object.

Information System — See **Automated Information System (AIS)**.

Legacy System — Any currently operating automated system that incorporates obsolete computer technology, such as proprietary hardware, closed systems, “stovepipe” design, or obsolete programming languages or **database** systems.

Life-Cycle Management (LCM) — A management process, applied throughout the life of an **AIS**, that bases all programmatic decisions on the anticipated mis-

	sion-related and economic benefits derived over the life of the AIS [DOD93a].
Message	Mechanism by which **objects** in an OO system request **services** of each other. Sometimes this is used as a synonym for **operation**.
Method	An operation upon an **object**, defined as part of the declaration of a **class**; all methods are **operation**s, but not all operations are methods [BOO94a].
Migration	The transition of support and operations of software functionality from a **legacy system** to a **migration system**.
Migration System	An existing **AIS**, or a planned and approved AIS, that has been officially designated to support standard processes for a functional activity applicable DoD-wide or DoD Component-wide [DOD93a]. Ordinarily, an AIS that has been designated to assume the functionality of a legacy AIS.
Monomorphism	A concept in type theory, according to which a name (such as a variable declaration) may only denote **objects** of the same **class** [BOO94a].
Object	A combination of state and a set of methods that explicitly embodies an **abstraction** characterized by the behavior of relevant requests. An **object** is an instance of an implementation and an interface. An object models a real-world entity (such as a person, place, thing, or concept), and it is implemented as a computational entity that encapsulates state and operations (internally implemented as data and methods) and responds to requestor services.
Object-Based Programming	A method of programming in which programs are organized as cooperative collections of **objects**, each of which represents an instance of some type, and whose types are all members of a hierarchy of types . . . somewhat constrained by the existence of static binding and **monomorphism** [BOO94a].

Object-Oriented Analysis	A method of analysis in which requirements are examined from the perspective of the **classes** and **objects** found in the vocabulary of the problem domain [BOO94a].
Object-Oriented Decomposition	The process of breaking a system into parts, each of which represents some **class** or **object** from the problem domain [BOO94a].
Object-Oriented Design	A method of design encompassing the process of OO decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design [BOO94a].
Object-Oriented Programming	A method of implementation in which programs are organized as cooperative collections of **objects**, each of which represents an instance of some **class**, and whose classes are members of a hierarchy of classes united via **inheritance** relationships. In such programs, classes are generally viewed as static, whereas objects typically have a much more dynamic nature, which is encouraged by the existence of dynamic binding and **polymorphism** [BOO94a].
Object-Oriented Technology (OOT)	OOT consists of a set of methodologies and tools for developing and maintaining software systems using software **objects** composed of encapsulated data and operations as the central paradigm.
Object Request Broker (ORB)	Program that provides a location and implementation-independent mechanism for passing a **message** from one **object** to another.
Operation	A specific behavior that an **object** exhibits, implemented as a procedure (or function) contained within the object.
Polymorphism	The same operation may behave differently on different classes [RUMB91].
Reengineering	The process of examining and altering an existing system to reconstitute it in a new form. May include reverse engineering (analyzing a system and producing a representation at a higher

level of **abstraction**, such as design from code), restructuring (transforming a system from one representation to another at the same level of abstraction), redocumentation (analyzing a system and producing user or support documentation), forward engineering (using software products derived from an existing system, together with new requirements, to produce a new system), retargeting (transforming a system to install it on a different target system), and translation (transforming source code from one language to another or from one version of a language to another) [DOD94a].

Requirement (1) Characteristic that a system or **CSCI** must possess in order to be acceptable to the acquirer. (2) A mandatory statement in this standard or another portion of the **contract** [DOD94a].

Responsibility A **contract** that a **class** must support, intended to convey a sense of the purpose of the class and its place in the system [HUTT94, p. 192].

Service A service is a specific behavior that an object is responsible for exhibiting [CDYD91].

Software Computer programs and computer databases. Note: Although some definitions of software includes documentation, MIL-STD-498 limits the scope of this term to computer programs and computer databases in accordance with Defense Federal Acquisition Regulation Supplement 227.401 [DOD94a].

Software Development A set of activities that results in **software** products. Software development may include new development, modification, reuse, **reengineering**, or any other activities that result in software products [DOD94a].

Software Engineering In general usage, a synonym for **software development**. As used in this standard [MIL-STD 498], a subset of software development consisting of all activities except qualification testing. The standard makes this distinction for the sole purpose of giving separate names to the software engineering and software test environments [DOD94a].

Software Engineering Environment	The facilities, hardware, software, firmware, procedures, and documentation needed to perform **software engineering**. Elements may include but are not limited to computer-aided software engineering (CASE) tools, compilers, assemblers, linkers, loaders, operating systems, debuggers, simulators, emulators, documentation tools, and **database** management systems [DOD94a].
Software System	A system consisting solely of software and possibly the computer equipment on which the software operates [DOD94a].
Weapon System	Items that can be used directly by the Armed Forces to carry out combat missions and that cost more than 100,000 dollars or for which the eventual total procurement cost is more than 10 million dollars. That term does not include commercial items sold in substantial quantities to the general public (Section 2403 of 10 U.S.C., reference (bb)) [DOD93a].

LIST OF ACRONYMS

AI	Artificial Intelligence
AIS	Automated Information Systems
AMO	At Most One
ANSI	American National Standards Institute
API	Application Programming Interface
ARPA	Advanced Research Projects Agency
BDE	Borland Database Engine
BLOB	Binary Large Object
BLSM	Base-Level System Modernization
C2I	Command, Control and Information
C3I	Command, Control, Communications, and Intelligence
C4I	Command, Control, Communications, Computers, and Intelligence
CAD	Computer-Aided Design
CASE	Computer-Aided Software Engineering
CIM	Corporate Information Management
COTS	Commercial off-the-Shelf
CORBA	Common Object Request Broker Architecture
CRC	Class-Responsibility-Collaborators
CY	Calendar Year
DBMS	Database Management Systems
DID	Data Item Description
DISA	Defense Information Systems Agency
DoD	Department of Defense
GCCS	Global Command and Control System

GIS	Geographical Information System
GUI	Graphical User Interface
IDA	Institute for Defense Analyses
IDAPI	Integration Database Application Program Interface
LCM	Life-Cycle Management
MDA	Milestone Decision Authority
NDI	Non-Developmental Item
ODBC	Open Database Connection
ODM	Object Data Management
ODMG	Object Data Management Group
OID	Object Identification
OMB	Office of Management and Budget
OMG	Object Management Group
OMT	Object Modeling Technique (Rumbaugh's OOT methodology)
OO	Object Oriented
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OODBMS	Object-Oriented Database Management System
OOP	Object-Oriented Programming
OOPSLA	Object-Oriented Programming Systems, Languages, and Applications
OOSE	Object-Oriented Software Engineering
OOT	Object-Oriented Technology
ORB	Object Request Broker
ORDBMS	Object Relational Database Management System
OSD	Office of the Secretary of Defense
PDSS	Post-Deployment Software Support
PM	Program Manager
RA	Requirements Analysis
RDA	Remote Data Access

RDBMS	Relational Database Management System
SOMA	Semantic Object Oriented Modeling Approach
SSS	System/Segment Specification
TAFIM	Technical Architecture Framework for Information Management